Chapter One
Overview and Introduction DAA

Outline

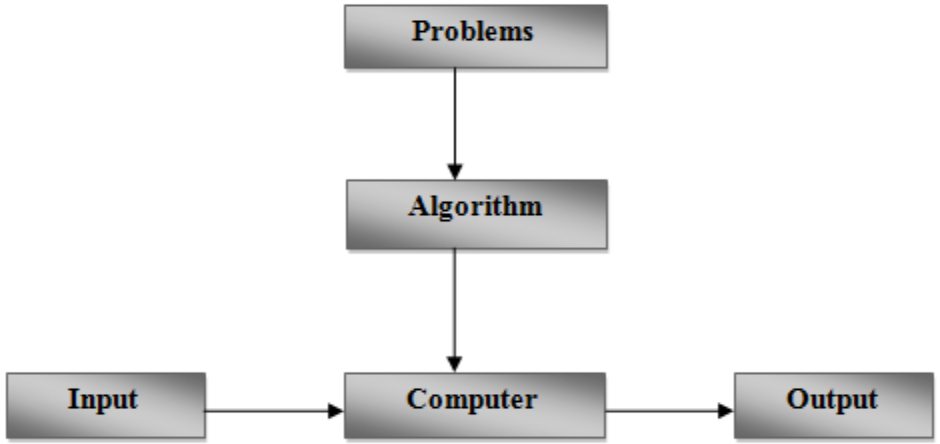
I. Introduction
II. Order notation
III. Analysis of algorithm
IV. Complexity

I. Introduction

- **What is Algorithm?**
  - It is a well-defined computational procedure that takes some value or a set of values as input and produces some value or a set of values as output.
  - it is a step- by- step procedure for solving a problem in a finite amount of time.
  - Algorithms are the dynamic part of a program's world model.
  - Unlike programs, algorithms are not dependent on a particular programming language, machine, system, or compiler

Cont..

- An algorithm transforms data structures from one state to another state in two ways:
  - An algorithm may change the value held by a data structure.
  - An algorithm may change the data structure itself.



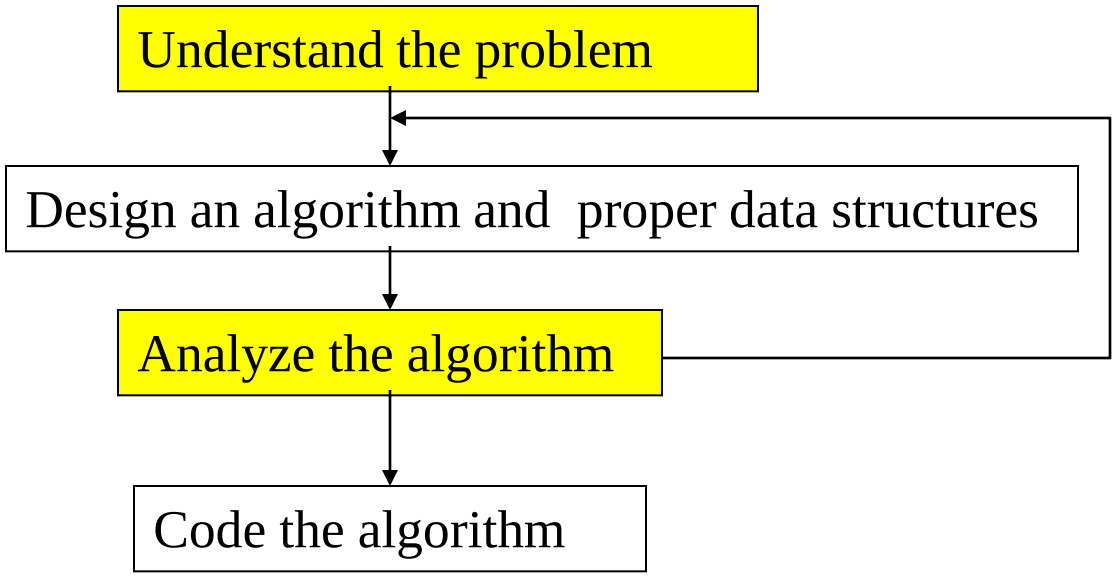
Properties of an algorithm

- **Finiteness**: Algorithm must complete after a finite number of steps.
- **Definiteness**: Each step must be clearly defined, having one and only one interpretation. At each point in computation, one should be able to tell exactly what happens next.
- **Sequence**: Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted.
- **Feasibility**: It must be possible to perform each instruction.
- **Correctness**: It must compute correct answer for all possible legal inputs

Cont..

- **Language Independence**: It must not depend on any one programming language.
- **Completeness**: It must solve the problem completely.
- **Effectiveness**: It must be possible to perform each step exactly and in a finite amount of time.
- **Efficiency**: It must solve with the least amount of computational resources such as time and space.
- **Generality**: Algorithm should be valid on all possible inputs.
- **Input/output**: There must be a specified number of input values, and one or more result values.

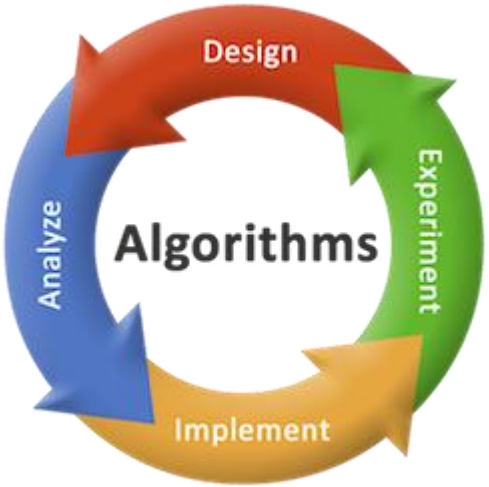
Algorithmic Problem Solving



- Ascertaining the Capabilities of a Computational Device
- Choosing between Exact and Approximate Problem Solving
- Deciding on Appropriate Data Structures
- Algorithm Design
- Algorithm Analysis
- Coding

Problem Development Steps

- The following steps are involved in solving computational problems.
  1. Problem definition
  2. Development of a model
  3. Specification of an Algorithm
  4. Designing an Algorithm
  5. Checking the correctness of an Algorithm
  6. Analysis of an Algorithm
  7. Implementation of an Algorithm
  8. Program testing
  9. Documentation



Algorithm Vs Pseudocode?

- A pseudocode is a high- level description of an algorithm in which it is
  - More structured than English text
  - Less detailed than a program
  - Preferred notation for describing algorithms
  - Hides program design issues
- An algorithm is a systematic, logical approach that provides a step-by-step procedure for computers to solve a specific problem. Pseudocode is a simplified version of programming codes, written in plain English language and used to outline a program before its implementation.

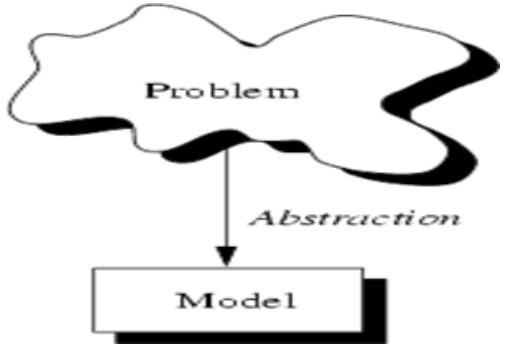
Algorithm Vs. program

- A program is written in order to solve a problem. A solution to a problem actually consists of two things:
    - A way to organize the data
    - Sequence of steps to solve the problem
- The way data are organized in a computer's memory is said to be Data Structure and the sequence of computational steps to solve a problem is said to be an algorithm.
- Therefore, a program is nothing but data structures plus algorithms.

Parameters	Algorithm	Pseudocode	Program
Meaning and Definition	An algorithm is a systematic, logical approach that provides a step-by-step procedure for computers to solve a specific problem.	Pseudocode is a simplified version of programming codes, written in plain English language and used to outline a program before its implementation.	A program is a set of instructions written in a computer language, which is then compiled or interpreted to be understood by the computer.
Expression and Use	Algorithms can be expressed using flowcharts, natural language, and other methods.	Pseudocode includes various control structures such as repeat-until, if-then-else, while, for, and case.	A program is written in a computer language and needs to be compiled or interpreted to be understood by the computer.

Data Structures

- Given a problem, the first step to solve the problem is obtaining one's own abstract view, or *model*, of the problem. This process of modeling is called abstraction



Abstraction is a process of classifying characteristics as relevant and irrelevant for the particular purpose at hand and ignoring the irrelevant ones.

- Is the structural representation of logical relationship between elements of data.
- It affects the design of both the structural and function aspect of the program.

The model defines an abstract view to the problem.

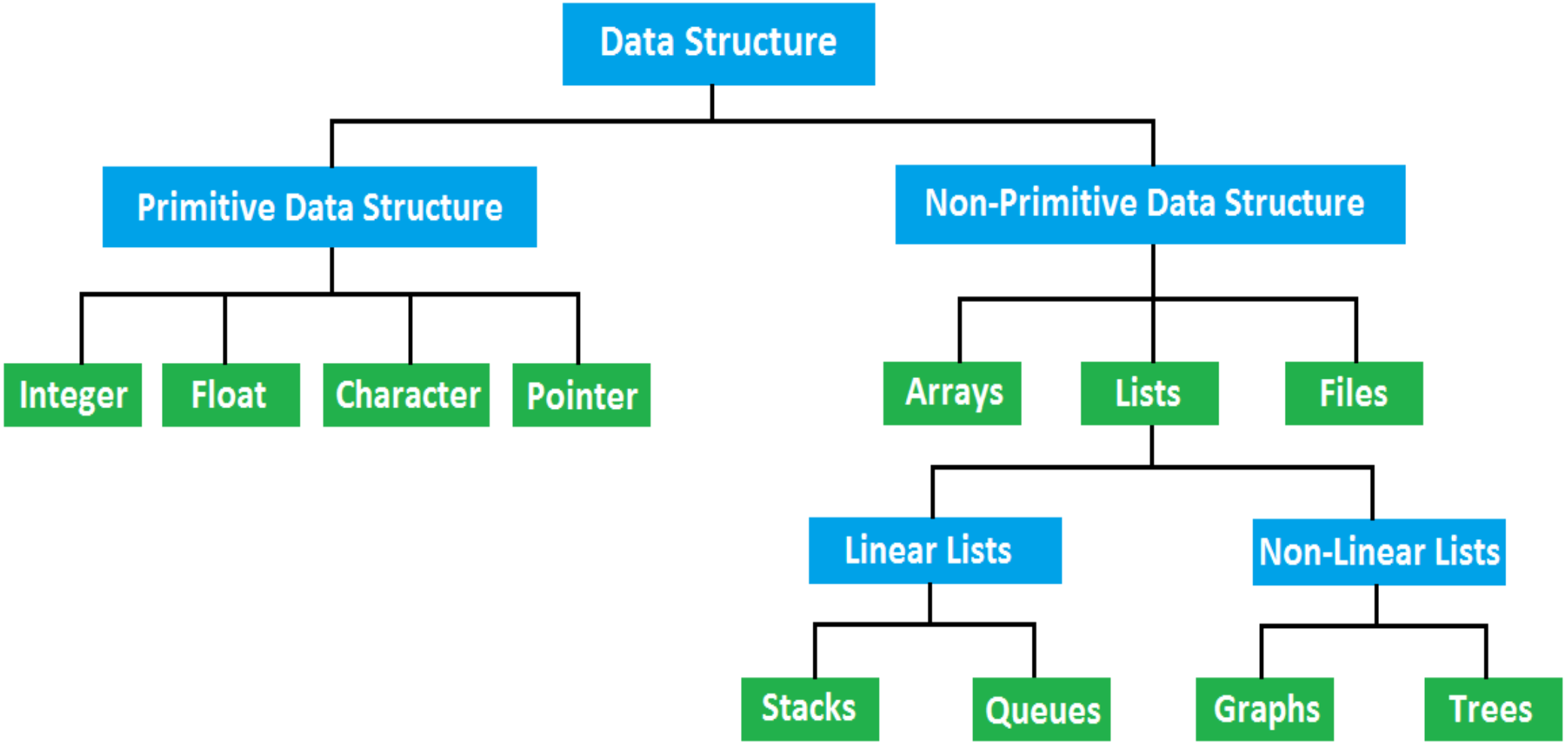
This implies that the model focuses only on problem related stuff and that a programmer tries to define the *properties* of the problem.

- These properties include
- The *data* which are affected and
- The *operations* that are involved in the problem.

Abstract Data Types

- An ADT consists of an abstract data structure and operations. Put in other terms, an ADT is an abstraction of a data structure.
- The ADT specifies:
  - What can be stored in the Abstract Data Type
  - What operations can be done on/by the Abstract Data Type.
- For example, if we are going to model employees of an organization:
- This ADT stores employees with their relevant attributes and discarding irrelevant attributes.
- This ADT supports hiring, firing, retiring … operations.
- A data structure is a language construct that the programmer has defined in order to implement an abstract data type.
- There are lots of formalized and standard Abstract data types such as Stacks, Queues, Trees, etc.

Classification of Data Structure



Complexity Analysis

- An objective way to evaluate the cost of an algorithm or code section.
- The cost is computed in terms of space or time, usually.
- The goal is to have a meaningful way to compare algorithms based on a common measure.
- Complexity analysis has two phases,
  - Algorithm analysis
  - Complexity analysis

Mathematical Analysis of Algorithms

- After developing pseudo-code for an algorithm, we wish to analyze its efficiency as a function of the size of the input, n in terms of how many times the elementary operation is performed. Here is a general strategy:
- Decide on a parameter(s) for the input, n.
- Identify the basic operation.
- Evaluate if the elementary operation depends only on n (otherwise evaluate best, worst, and average-case separately.
- Set up a summation corresponding to the number of elementary operations
- Simplify the equation to get as simple of a function T(n) as possible

Performance Analysis

- What is Performance Analysis of an algorithm?
- If we want to go from “Dilla " to “Hawassa", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us.
- Similarly, in computer science, there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one.
- Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.
- When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements. The formal definition is as follows.

Cont….

- Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.
- That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.
- We compare algorithms with each other which are solving the same problem, to select the best algorithm.
- To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.

Cont….

- Generally, the performance of an algorithm depends on the following elements.
    1. Whether that algorithm is providing the exact solution for the problem?
    2. Whether it is easy to understand?
    3. Whether it is easy to implement?
    4. How much space (memory) it requires to solve the problem?
    5. How much time it takes to solve the problem? Etc.
- When we want to analyze an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements.
- Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

Cont….

- Performance analysis of an algorithm is performed by using the following measures
  - Space required to complete the task of that algorithm **(Space Complexity).** It includes program space and data space
  - Time required to complete the task of that algorithm **(Time Complexity).**

  What is Space complexity?
- When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes.
  1. To store program instructions.
  2. To store constant values.
  3. To store variable values.
  4. And for few other things like function calls, jumping statements etc.

Cont....

- Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.
- To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C++ Programming Language compiler requires the following.

  1.2 bytes to store Integer value.

  2.4 bytes to store Floating Point value.

  3.1 byte to store Character value.

  4.6 (OR) 8 bytes to store double value

Time complexity

- **What is Time complexity?**
- Every algorithm requires some amount of computer time to execute its instruction to perform the task, is called time complexity
- The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution. Generally, the running time of an algorithm depends upon the following.
  - Whether it is running on **Single** processor machine or **Multi** processor machine.
  - Whether it is a **32 bit** machine or **64 bit** machine.
  - **Read** and **Write** speed of the machine.
  - The amount of time required by an algorithm to perform **Arithmetic** operations, **logical** operations, **return** value and **assignment** operations etc.
  - **Input** data

Algorithm Analysis

- Algorithm analysis requires a set of rules to determine how operations are to be counted.
- There is no generally accepted set of rules for algorithm analysis.
- In some cases, an exact count of operations is desired; in other cases, a general approximation is sufficient.
- The rules presented that follow are typical of those intended to produce an exact count of operations.

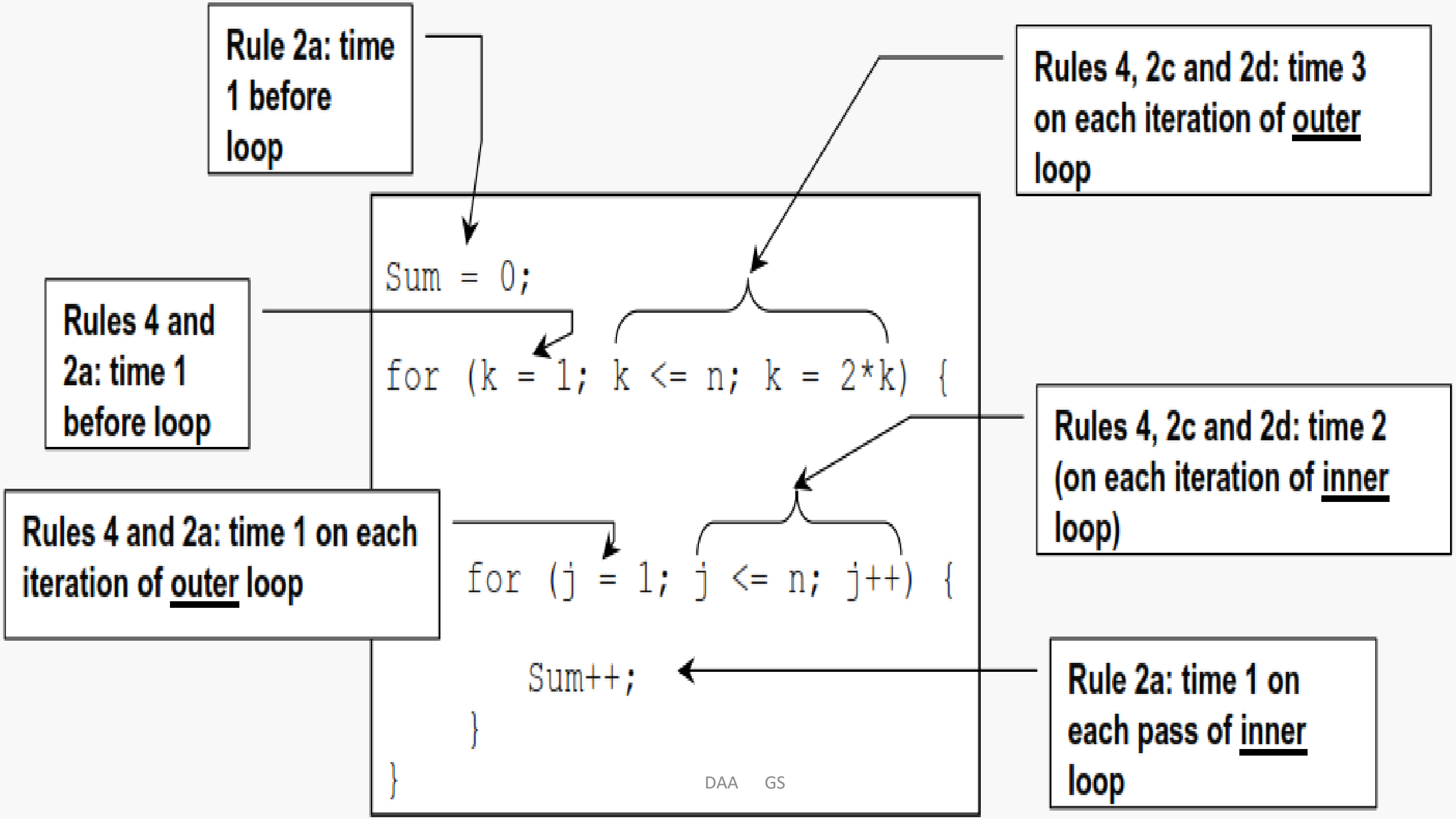
Rules

1. We assume an arbitrary time unit.
2. Execution of one of the following operations takes time 1:
   A. assignment operation
   B. single I/O operations
   C. single Boolean operations, numeric comparisons
   D. single arithmetic operations
   E. function return
   F. array index operations, pointer dereferences

More Rules

3. Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection.
4. Loop execution time is the sum, over the number of times the loop is executed, of the body time + time for the loop check and update operations, + time for the loop setup.

   Always assume that the loop executes the maximum number of iterations possible
5. Running time of a function call is 1 for setup + the time for any parameter calculations + the time required for the execution of the function body.


Rule 2a: time 1 before loop
Rules 4, 2c and 2d: time 3 on each iteration of outer loop
Rules 4 and 2a: time 1 before loop
Rules 4, 2c and 2d: time 2 (on each iteration of inner loop)
Rules 4 and 2a: time 1 on each iteration of outer loop
Rule 2a: time 1 on each pass of inner loop
Sum = 0;
for (k = 1; k <= n; k = 2*k) {
for (j = 1; j <= n; j++) {
Sum++;
}
}

Cont….

- When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent.
- We check only, how our program is behaving for the different input values to perform all the operations like Arithmetic, Logical, Return value and Assignment etc.

	Cost Time require for line (Units)	Repeatation No. of Times Executed	Total Total Time required in worst case
int sumOfList(int A[ ], int n)			
{			
int sum = 0, i;	1	1	1
for(i = 0; i < n; i++)	1 + 1 + 1	1 + (n+1) + n	2n + 2
sum = sum + A[i];	2	n	2n
return sum;	1	1	1
}			
			4n + 4 Total Time required

Kinds of Analysis

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are.
- Best case
  - Provides a lower bound on running time
  - Input is the one for which the algorithm runs the fastest

$$Lower\ Bound \leq Running\ Time \leq Upper\ Bound$$

- Average case
  - Provides a prediction about the running time
  - Assumes that the input is random

Asymptotic Notation/Growth of Functions

$O \approx \leq$
$\Omega \approx \geq$
$\Theta \approx =$
$o \approx <$
$\omega \approx >$

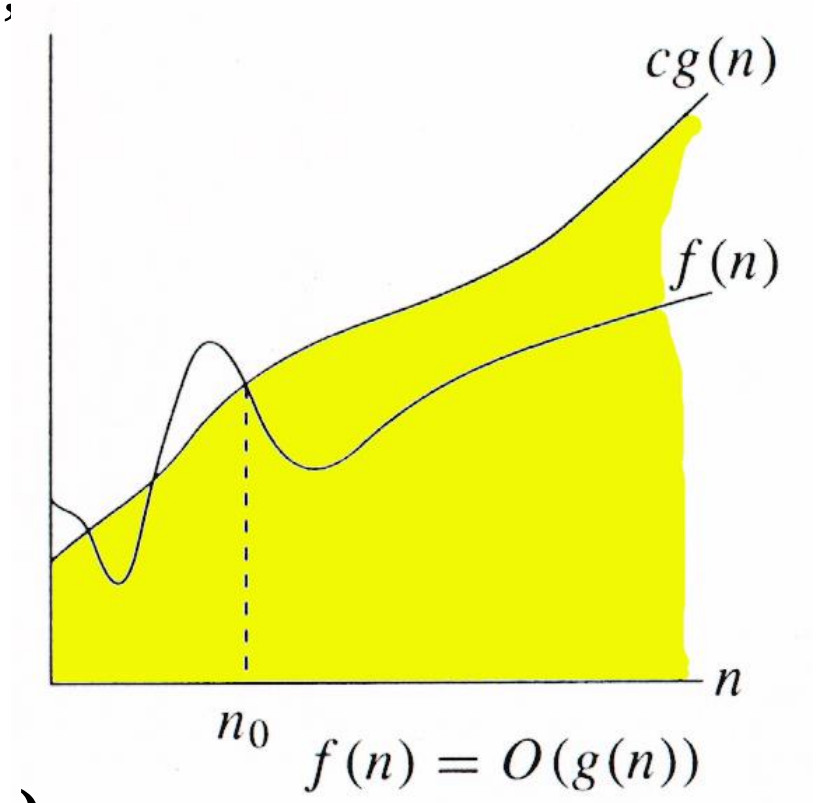
- **Order of growth** is how the time of execution depends on the length of the input. Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm.
- But when we calculate the complexity of an algorithm it does not provide the exact amount of resource required.
- So instead of taking the exact amount of resource, we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm.
- ***O, Ω, Θ o, ω***
- In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity can be Space Complexity or Time Complexity).

O-notation

For function $g(n)$, we define $O(g(n))$, big-O of n, as the set:

$f(n) = O(g(n))$
∃ positive constants c and n_0,
such that $\forall n \geq n_0$,
we have $f(n) \leq cg(n)$ }

Intuitively: Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.



$g(n)$ is an *asymptotic upper bound* for $f(n)$.
always indicates the maximum time required by an algorithm for all input values.

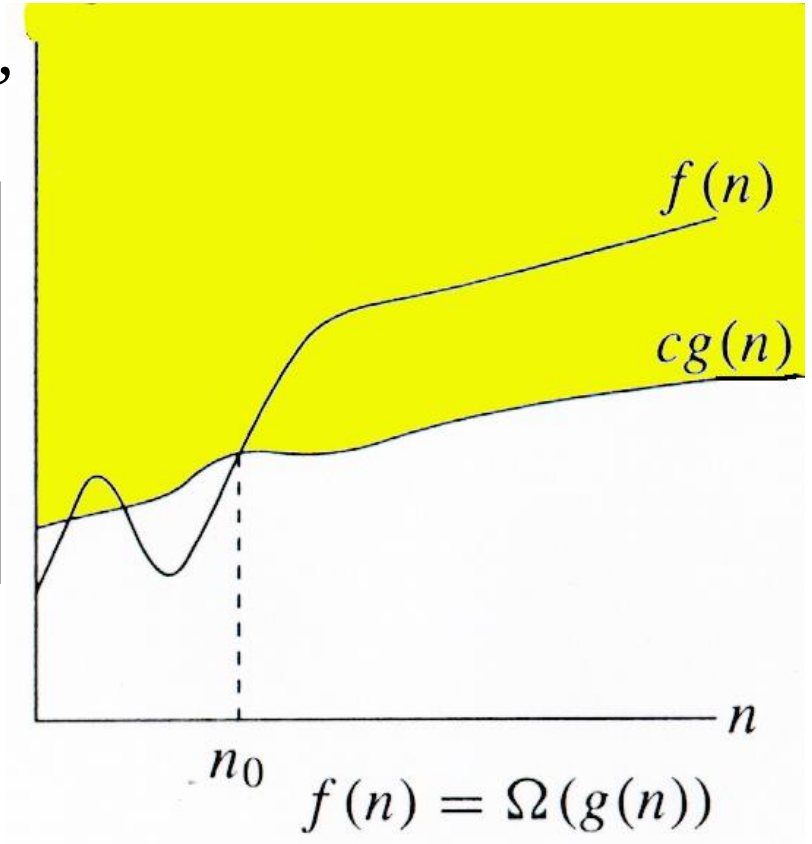
Ω -notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

$f(n) = \Omega(g(n))$
$\exists$ positive constants c and n_0,
such that $\forall n \geq n_0$,

we have $f(n) \geq cg(n)\}$

Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.



$g(n)$ is an *asymptotic lower bound* for $f(n)$.
always indicates the minimum time required by an algorithm for all input values

Θ-notation

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

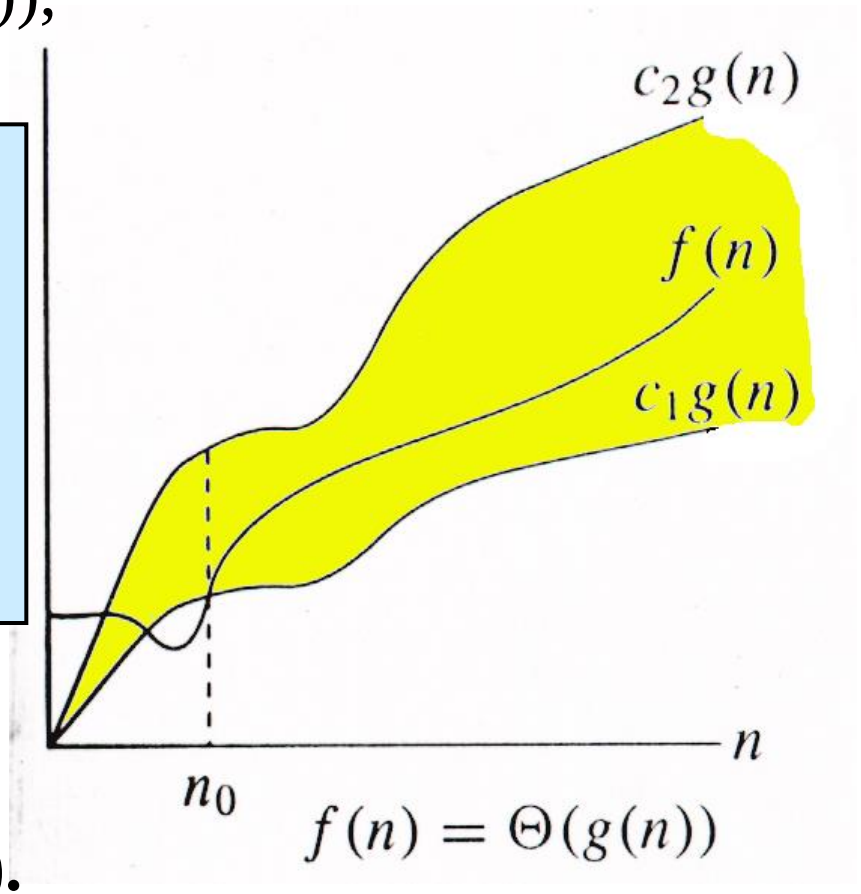
$f(n) = \Theta(g(n))$

∃ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$,

we have $c_1 g(n) \leq f(n) \leq c_2 g(n)$

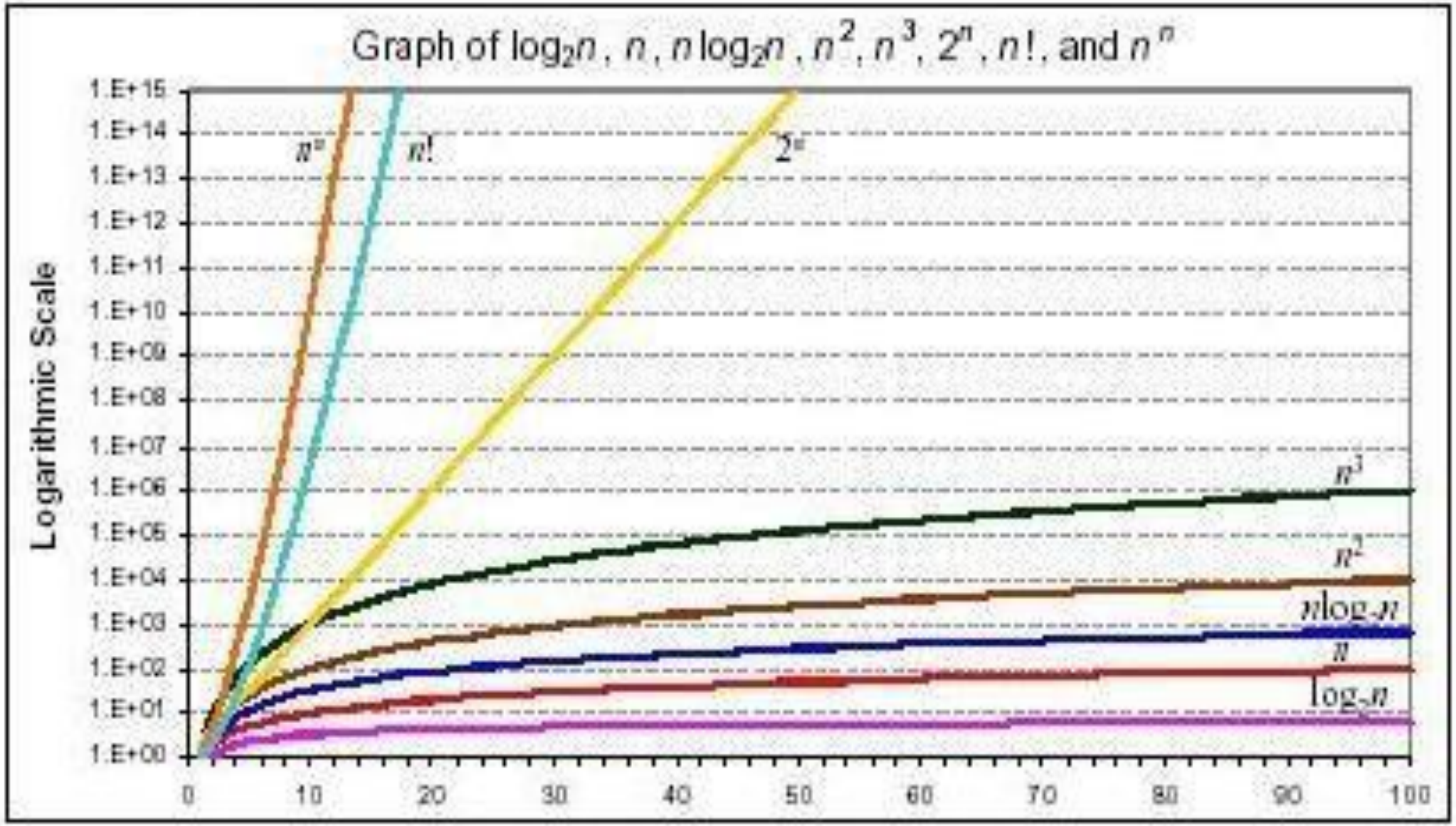
}

Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.

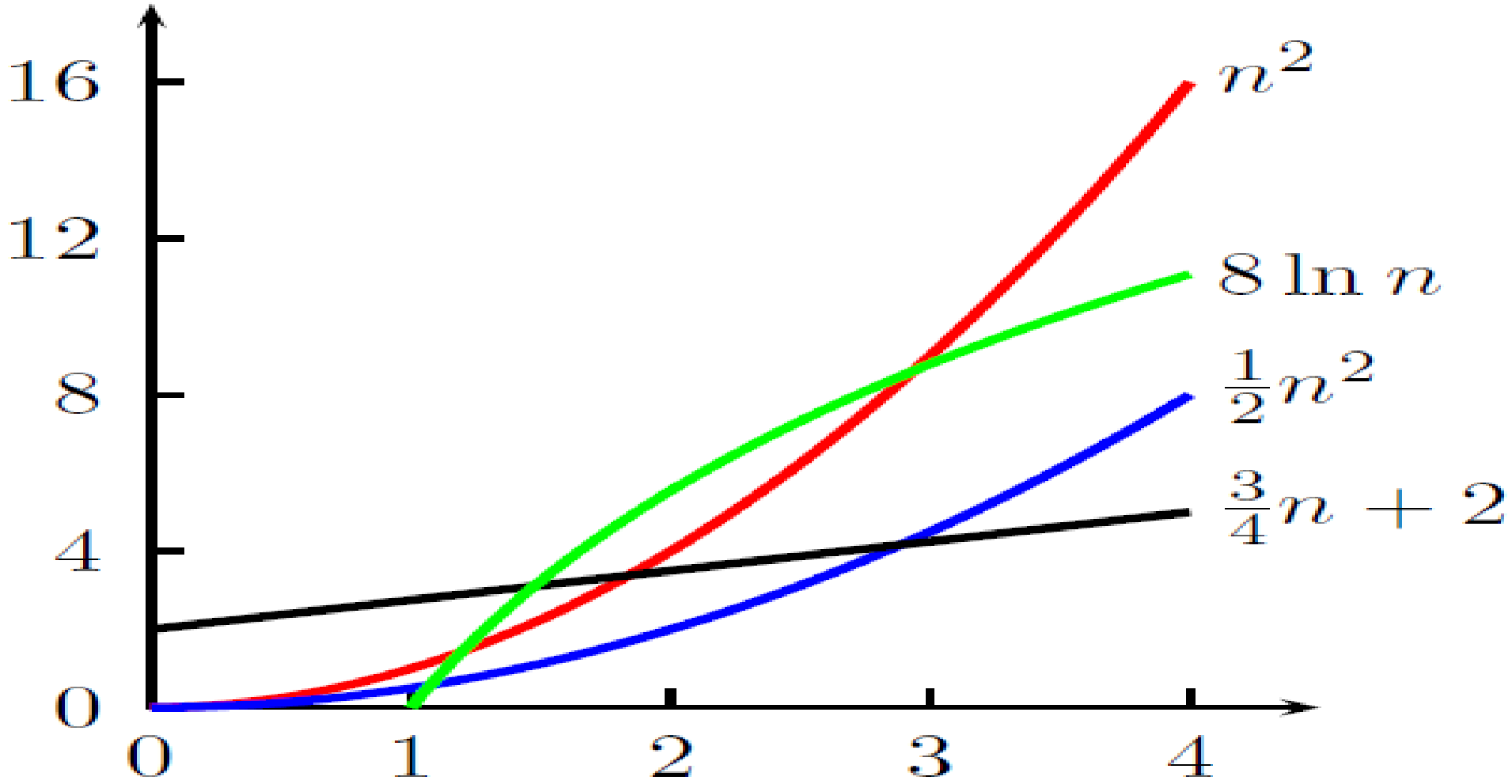


$g(n)$ is an *asymptotically tight bound* for $f(n)$.

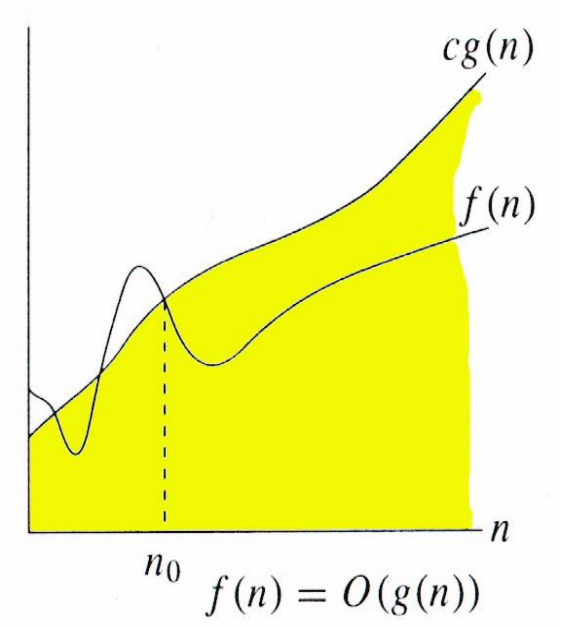
always indicates the average time required by an algorithm for all input values.

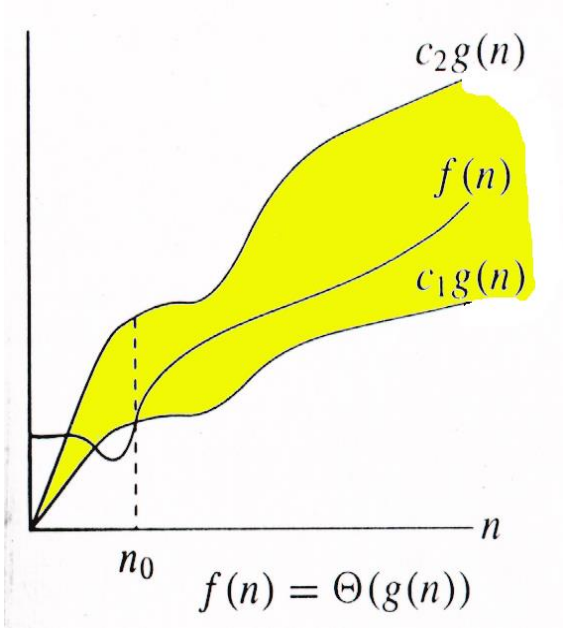

Graph of $\log_2 n$, n, $n \log_2 n$, n^2, n^3, 2^n, $n!$, and n^n
Logarithmic Scale
1.E+15
1.E+14
1.E+13
1.E+12
1.E+11
1.E+10
1.E+09
1.E+08
1.E+07
1.E+06
1.E+05
1.E+04
1.E+03
1.E+02
1.E+01
1.E+00
0
10
20
30
40
50
60
70
80
90
100
n^n
$n!$
2^n
n^3
n^2
$n\log n$
n
$\log n$

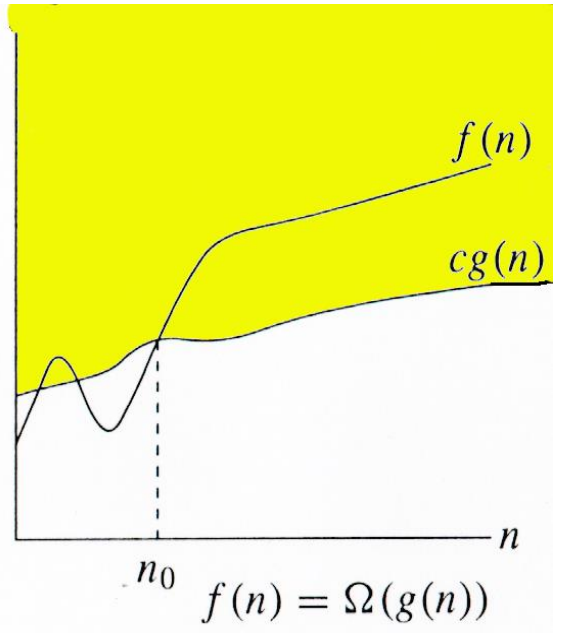
Example O(n^2)

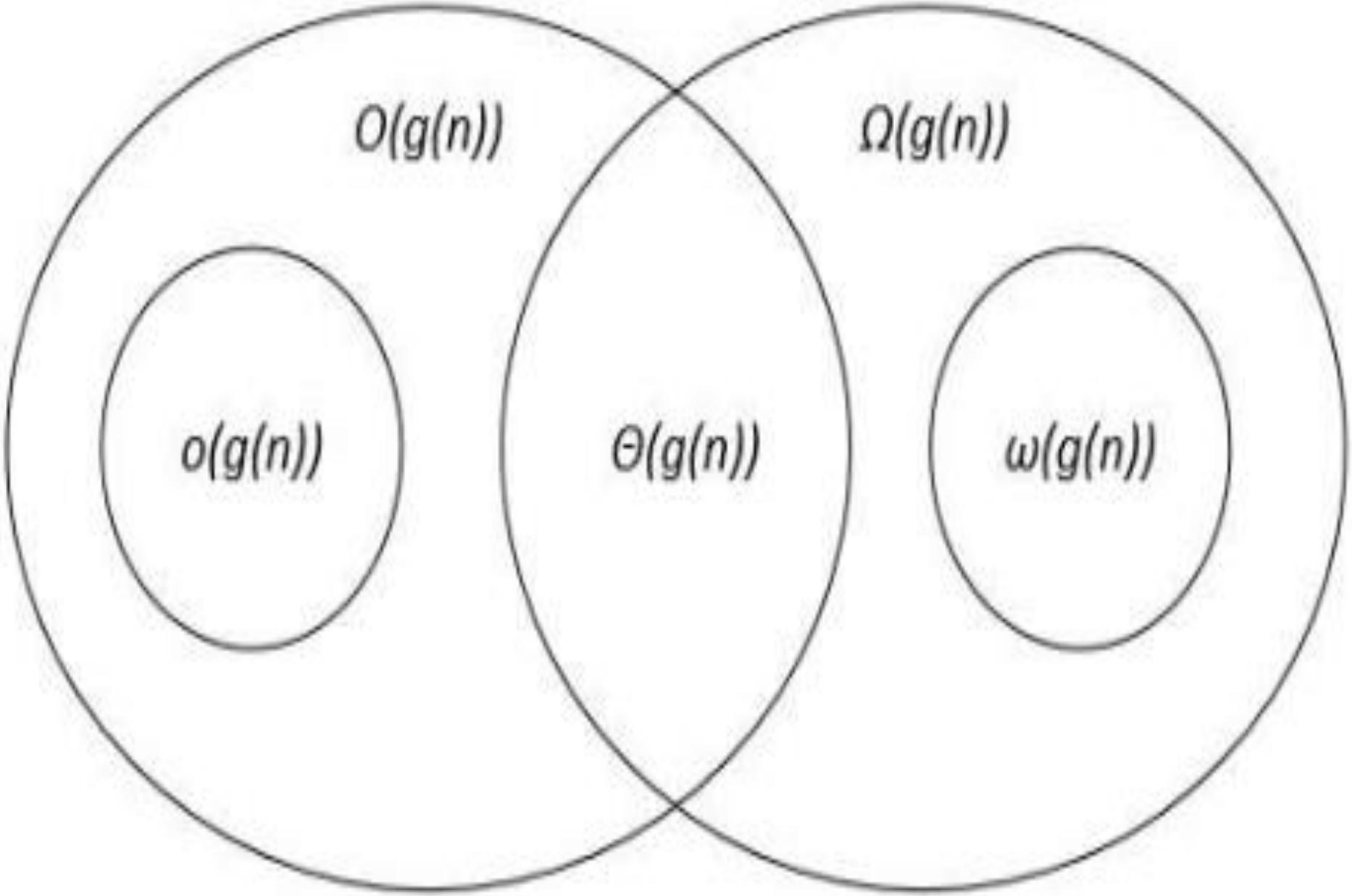


Relations Between *O*, Θ, Ω, o, ω








O(g(n))
Ω(g(n))
o(g(n))
Θ(g(n))
ω(g(n))

Relations Between Θ, Ω, O

Theorem : For any two functions $g(n)$ and $f(n)$,

$f(n) = \Theta(g(n))$ iff

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

- I.e., $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

- In practice, asymptotically tight bounds are obtained from asymptotic upper and lower bounds.

Running Time

• The running time depends on the input: an already sorted sequence is easier to sort.
• Parameterize the running time by the size of the input, since short sequences are easier to sort than long ones.
• Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

Worst-case: (usually)

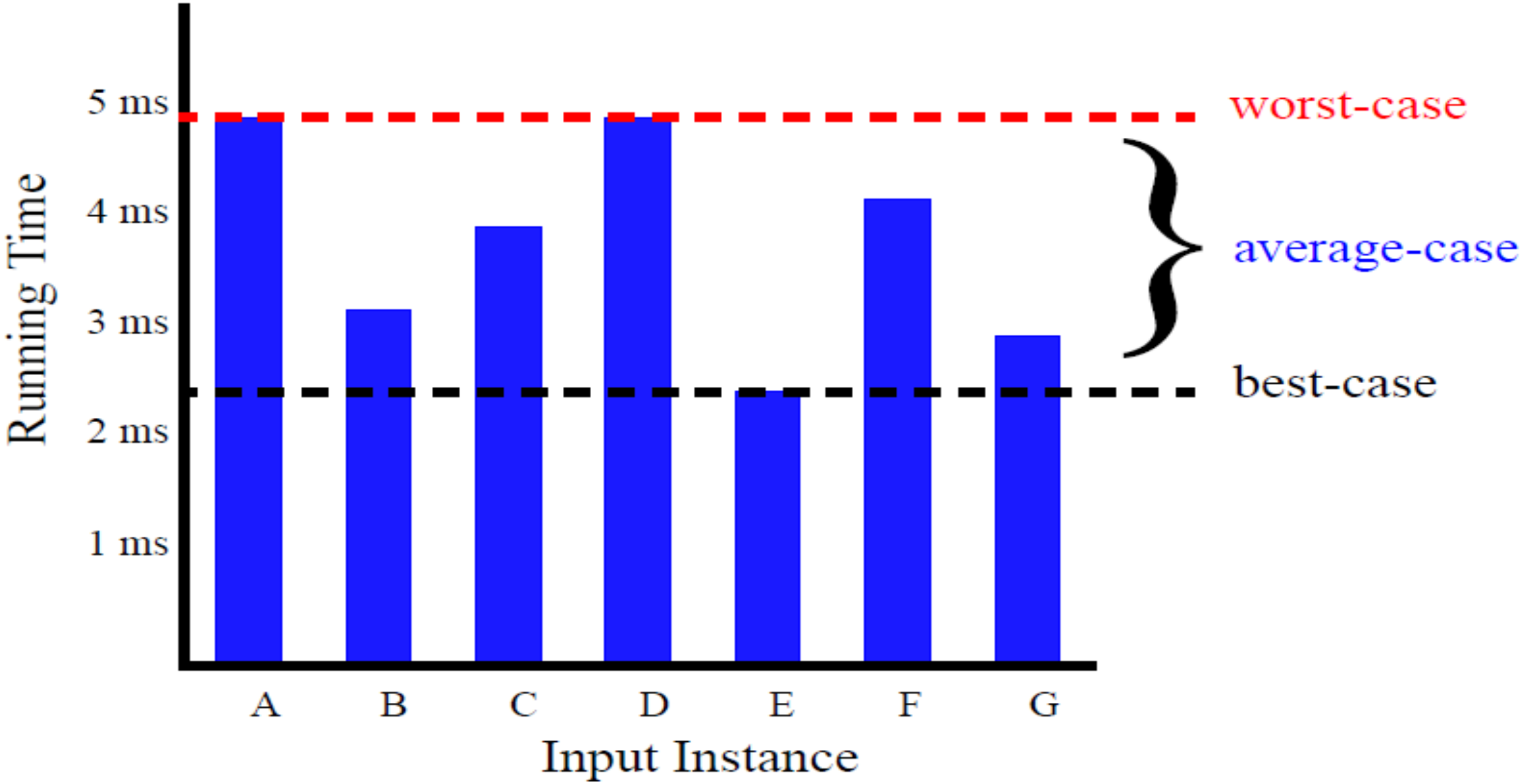
- T(n) = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- T(n) = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case:

- Cheat with a slow algorithm that works fast on some input.


5 ms
4 ms
3 ms
2 ms
1 ms
Running Time
worst-case
average-case
best-case
A
B
C
D
E
F
G
Input Instance

Analyzing Algorithms

operations	constant time
Consecutive stmts	sum of times
Conditionals	larger branch plus test
Loops	sum of iterations
Function calls	cost of function body
Recursive functions	solve recursive equation

Properties of asymptotic notation

- **Transitivity**

$$f(n) = \Theta(g(n)) \;\&\; g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \;\&\; g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \;\&\; g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$
$$f(n) = o\,(g(n)) \;\&\; g(n) = o\,(h(n)) \Rightarrow f(n) = o\,(h(n))$$
$$f(n) = \omega(g(n)) \;\&\; g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$$

- **Reflexivity**

$$f(n) = \Theta(f(n))$$
$$f(n) = O(f(n))$$
$$f(n) = \Omega(f(n))$$

Properties of Asymptotic Analysis

- **Symmetry**

$$f(n) = \Theta(g(n)) \text{ iff } g(n) = \Theta(f(n))$$

- **Complementarity**

$$f(n) = O(g(n)) \text{ iff } g(n) = \Omega(f(n))$$

$$f(n) = o(g(n)) \text{ iff } g(n) = \omega((f(n))$$

Exponentials

- **Useful Identities:**

$$a^{-1} = \frac{1}{a}$$

$$(a^m)^n = a^{mn}$$

$$a^m a^n = a^{m+n}$$

- **Exponentials and polynomials**

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0$$

$$\Rightarrow n^b = o(a^n)$$

Logarithms

$x = \log_b a$ is the exponent for $a = b^x$.

Natural log: $\ln a = \log_e a$

Binary log: $\lg a = \log_2 a$

$\lg^2 a = (\lg a)^2$

$\lg \lg a = \lg (\lg a)$

$$a = b^{\log_b a}$$

$$\log_c (ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b (1/a) = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

The Best Never Rest!