كلية معتمدة من الهيئة القومية
لضمان جودة التعليم والاعتماد

Assiut University

Course Title: Data Structures and Algorithms
Course Code: CS211

Prof. Dr. Khaled F. Hussain

Reference

- Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser, “Data Structures and Algorithms in Python”

Introduction

- We are interested in the design of “good” data structures and algorithms. Simply put, a ***data structure*** is a systematic way of organizing and accessing data, and an ***algorithm*** is a step-by-step procedure for performing some task in a finite amount of time.
- We would like to focus on the relationship between the running time of an algorithm and the size of its input. We are interested in characterizing an algorithm’s running time as a function of the input size.

Experimental Studies

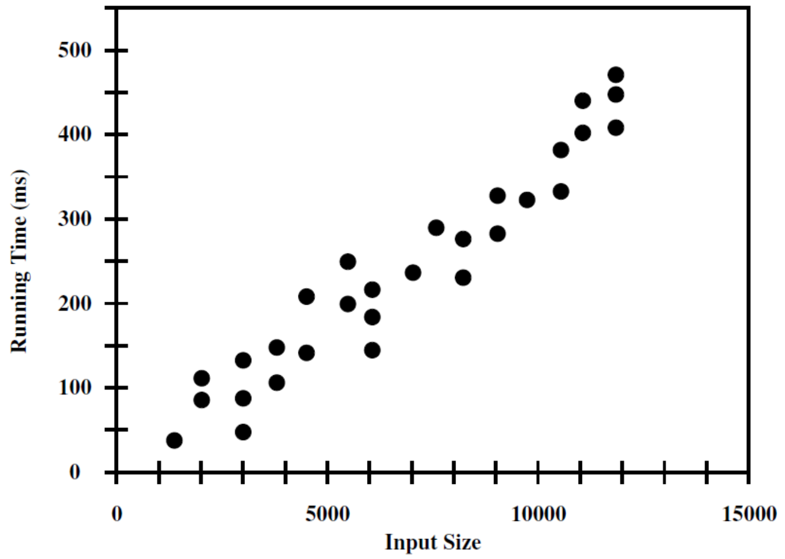
- If an algorithm has been implemented, we can study its running time by executing it on various test inputs and recording the time spent during each execution.

```
from time import time
start_time = time( ) # record the starting time
x=6  # algorithm
end_time = time( ) # record the ending time
elapsed = end_time-start_time # compute the elapsed time
print(elapsed)
```

- This code reports the number of seconds, or fractions

Experimental Studies (Cont.)

- The time function measures relative to what is known as the "wall clock." Because many processes share use of a computer's ***central processing unit*** (or ***CPU***), the elapsed time will depend on what other processes are running on the computer when the test is performed.
- A fairer metric is the number of CPU cycles that are used by the algorithm.
- We should perform independent experiments on many different test inputs of various sizes. We can then visualize the results by plotting the performance of each run of the algorithm as a point with *x*-coordinate equal to the input size, *n*, and *y*-coordinate equal to the running time, *t*.



Challenges of Experimental Analysis

- Experimental running times of two algorithms are difficult to directly compare unless the experiments are performed in the same hardware and software environments.
- Experiments can be done only on a limited set of test inputs.
- An algorithm must be fully implemented in order to execute it to study its running time experimentally.

This last requirement is the most serious drawback to the use of experimental studies.

Moving Beyond Experimental Analysis

- Our goal is to develop an approach to analyzing the efficiency of algorithms that:
    1. Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.
    2. Is performed by studying a high-level description of the algorithm without need for implementation.
    3. Takes into account all possible inputs.

Moving Beyond Experimental Analysis (Cont.)

- We will count how many primitive operations are executed, and use this number *t* as a measure of the running time of the algorithm.

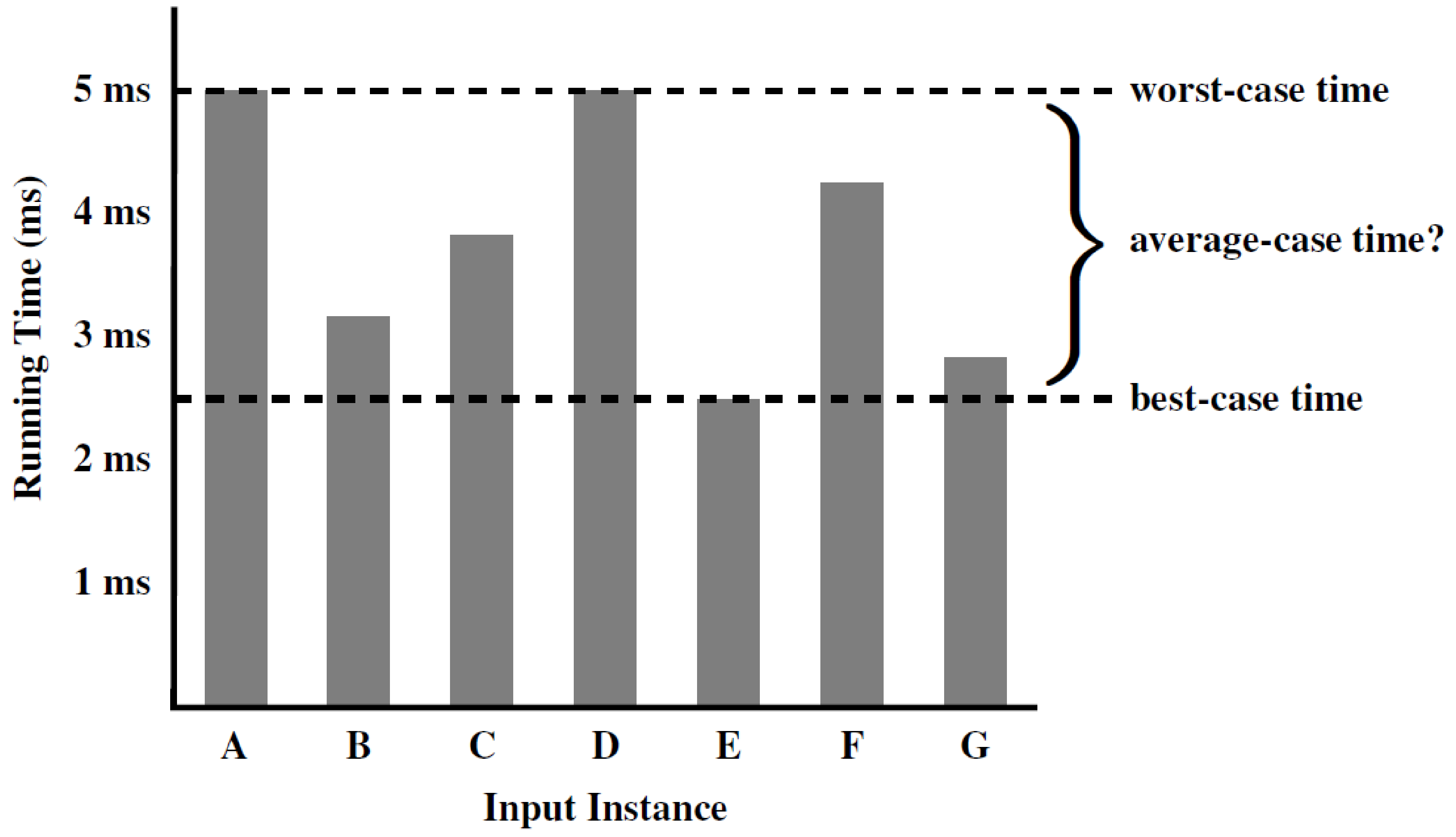
- We define a set of ***primitive operations*** such as the following:
    - Assigning an identifier to an object
    - Performing an arithmetic operation (for example, adding two numbers)
    - Comparing two numbers
    - ….

- The implicit assumption in this approach is that the running times of different primitive operations will be fairly similar.

Measuring Operations as a Function of Input Size

- To capture the order of growth of an algorithm’s running time, we will associate, with each algorithm, a function $f(n)$ that characterizes the number of primitive operations that are performed as a function of the input size n.

Worst-Case, best-case, and average-case



Worst-Case, best-case, and average-case (Cont.)

- An average-case analysis usually requires that we calculate expected running times based on a given input distribution, which usually involves sophisticated probability theory.
- Therefore, unless we specify otherwise, we will characterize running times in terms of the ***worst case***, as a function of the input size, *n*, of the algorithm.
- Making the standard of success for an algorithm to perform well in the worst case necessarily requires that it will do well on ***every*** input.

The seven fundamental functions

- We will briefly discuss the seven most important functions used in the analysis of algorithms.

1. The Constant Function

   $f(n) = c$, for some fixed constant c

   - The constant function is useful in algorithm analysis, because it characterizes the number of steps needed to do a basic operation.

2. The Logarithm Function

   $f(n) = \log_b n$, for some constant $b > 1$.

   - This function is defined as follows: $x = \log_b n$ if and only if $b^x = n$.
   - By definition, $\log_b 1 = 0$. The value b is known as the ***base*** of the logarithm.
   - We will typically omit it from the notation when it is 2. That is, for us, $\log n = \log_2 n$.
   - We note that most handheld calculators have a button marked LOG, but this is typically for calculating the logarithm base-10, not base-two.

The seven fundamental functions (Cont.)

2- The Logarithm Function

- Logarithm Rules: Given real numbers $a > 0$, $b > 1$, $c > 0$ and $d > 1$, we have:
  a) $\log_b(ac) = \log_b a + \log_b c$
  b) $\log_b(a/c) = \log_b a - \log_b c$
  c) $\log_b(a^c) = c \log_b a$
  d) $\log_b a = \log_d a / \log_d b$
  e) $b^{\log d\, a} = a^{\log d\, b}$

  - The notation $\log n^c$ denotes the value $\log(n^c)$.
  - The notation $\log^c n$, to denote the quantity, $(\log n)^c$

The seven fundamental functions (Cont.)

2- The Logarithm Function

- Examples
  - $\log(2n) = \log 2 + \log n = 1 + \log n$
  - $\log(n/2) = \log n - \log 2 = \log n - 1$
  - $\log n^3 = 3\log n$
  - $\log 2^n = n\log 2 = n \cdot 1 = n$
  - $\log_4 n = (\log n)/\log 4 = (\log n)/2$
  - $2^{\log n} = n^{\log 2} = n^1 = n$

The seven fundamental functions (Cont.)

3. The Linear Function

$$f(n) = n.$$

- The linear function also represents the best running time we can hope to achieve for any algorithm that processes each of n objects that are not already in the computer's memory, because reading in the n objects already requires n operations.

4. The N-Log-N Function

$$f(n) = n\log n,$$

- This function grows a little more rapidly than the linear function and a lot less rapidly than the quadratic function.
- For example, the fastest possible algorithms for sorting n arbitrary values require time proportional to $n\log n$.

The seven fundamental functions (Cont.)
5- The Quadratic Function

5. The Quadratic Function

$$f(n) = n^2.$$

- The main reason why the quadratic function appears in the analysis of algorithms is that there are many algorithms that have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. Thus, in such cases, the algorithm performs $n \cdot n = n^2$ operations.
- The quadratic function can also arise in the context of nested loops where the first iteration of a loop uses one operation, the second uses two operations, the third uses three operations, and so on. That is, the number of operations is $1+2+3+\cdots+(n-2)+(n-1)+n$.
- Proposition: For any integer $n \geq 1$, we have: $1+2+3+\cdots+(n-2)+(n-1)+n = n(n+1)/2$

The seven fundamental functions (Cont.)

6. The Cubic Function and Other Polynomials

6. The Cubic Function and Other Polynomials
   - The ***cubic function***, $f(n) = n^3$

Most of the functions we have listed so far can each be viewed as being part of a larger class of functions, the polynomials. A polynomial function has the form,

$$f(n) = a_0 + {}_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d$$

- Integer d, which indicates the highest power in the polynomial, is called the ***degree*** of the polynomial.

The seven fundamental functions (Cont.)
7. The Exponential Function

- Another function used in the analysis of algorithms is the ***exponential function***,

$$f(n) = b^n,$$

where b is a positive constant, called the ***base***, and the argument n is the ***exponent***.

- (Exponent Rules): Given positive integers a, b, and c, we have:

$$(b^a)^c = b^{ac}$$

$$b^a b^c = b^{a+c}$$

$$b^a / b^c = b^{a-c}$$

Geometric Sums

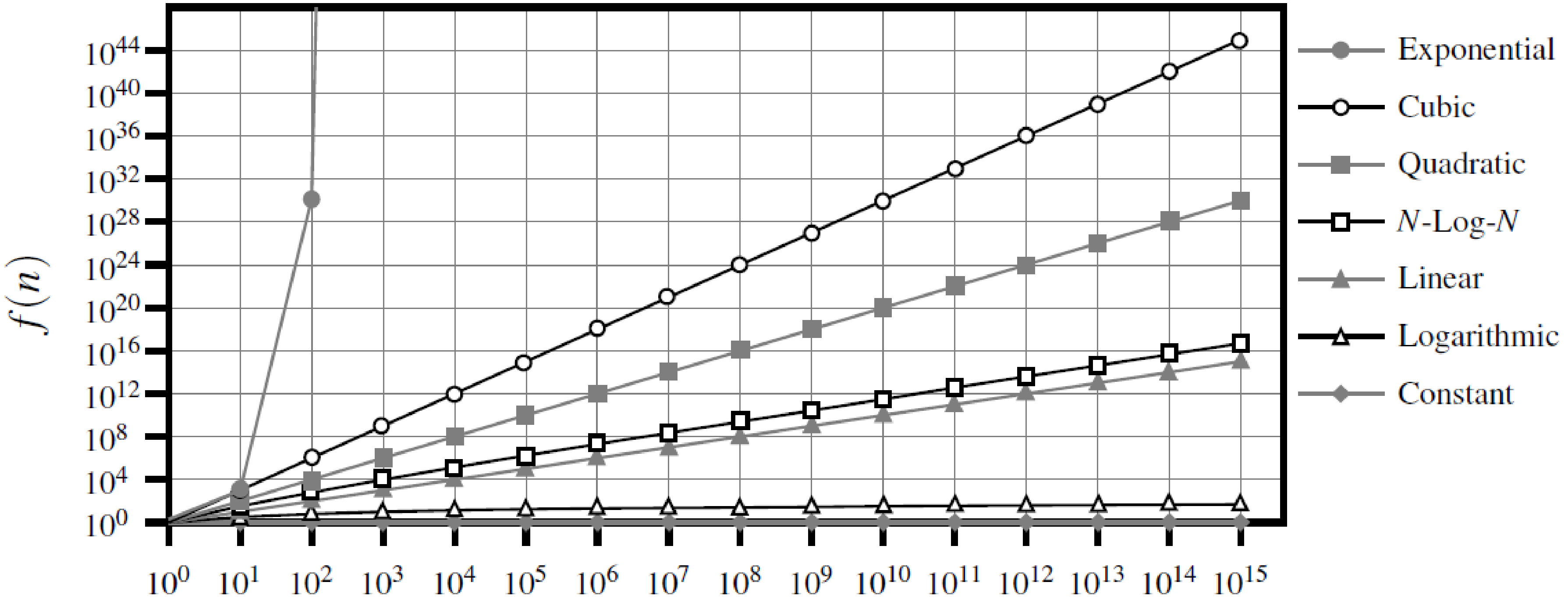
- For any integer $n \geq 0$ and any real number a such that $a > 0$ and $a \neq 1$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$$

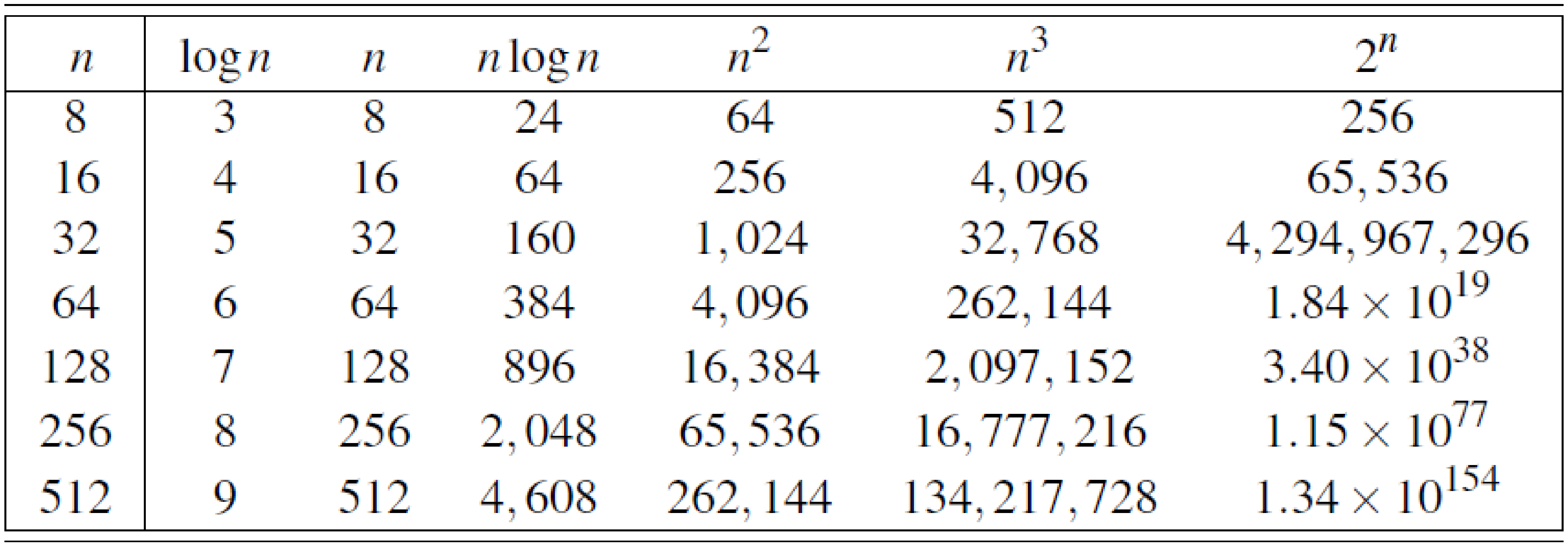
- For example, $1+2+4+8+\cdot\cdot\cdot+2^{n-1} = 2^n - 1$

Comparing Growth Rates

constant	logarithm	linear	*n*-log-*n*	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n



Comparing Growth Rates (Cont.)



n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Asymptotic Analysis

- Asymptotic analysis is a method of describing the limiting behavior of a function or expression, especially as it approaches infinity or some other limiting value.

- In algorithm analysis, we focus on the growth rate of the running time as a function of the input size *n.*

- It is often enough just to know that the running time of an algorithm ***grows proportionally to*** *n*.

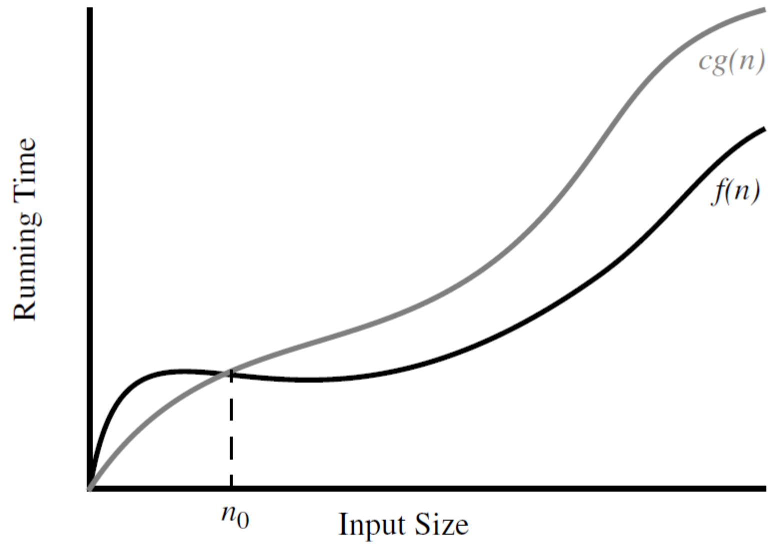
Example: finding the largest element of a Python list

```
def find max(data):
    biggest = data[0]          # The initial value to beat
    for val in data:           # For each value:
        if val > biggest       # if it is greater than the best so far,
            biggest = val      # we have found a new best (so far)
    return biggest             # When loop ends, biggest is the max
```

- This is a classic example of an algorithm with a running time that grows proportional to *n*.

The “Big-Oh” Notation

- Let *f* (*n*) and *g*(*n*) be functions mapping positive integers to positive real numbers. We say that *f* (*n*) is *O*(*g*(*n*)) if there is a real constant $c > 0$ and an integer constant *n*0 ≥ 1 such that *f* (*n*) ≤ *cg*(*n*), for *n* ≥ *n*0.
- This definition is often referred to as the “big-Oh” notation, for it is sometimes pronounced as “ *f* (*n*) is ***big-Oh*** of *g*(*n*).”
- “ *f* (*n*) ***is*** *O*(*g*(*n*)).” Alternatively, we can say “ *f* (*n*) is ***order of*** *g*(*n*).”



Example

- The function $8n+5$ is $O(n)$.

- By the big-Oh definition, we need to find a real constant $c>0$ and an integer constant $n0 \geq 1$ such that $8n+5 \leq cn$ for every integer $n \geq n0$. It is easy to see that a possible choice is $c = 9$ and $n0 = 5$. Indeed, this is one of infinitely many choices available because there is a trade-off between c and $n0$. For example, we could rely on constants $c = 13$ and $n0 = 1$.

- The algorithm, find max, for computing the maximum element of a list of n numbers, runs in $O(n)$ time.
- Justification:
  - The initialization before the loop begins requires only a constant number of primitive operations. Each iteration of the loop also requires only a constant number of primitive operations, and the loop executes n times.
  - Therefore, we account for the number of primitive operations being $c' + c'' \cdot n$ for appropriate constants c' and c'' that reflect, respectively, the work performed during initialization and the loop body. Because each primitive operation runs in constant time, we have that the running time of algorithm find max on an input of size n is at most a constant times n; that is, we conclude that the running time of algorithm find max is $O(n)$.

Some Properties of the Big-Oh Notation

- The big-Oh notation allows us to ignore constant factors and lower-order terms and focus on the main components of a function that affect its growth.
- Example: $5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$.
    - Justification: Note that $5n^4+3n^3+2n^2+4n+1 \leq (5+3+2+4+1)n^4 = cn^4$, for $c = 15$, when $n \geq n0 = 1$.

- Proposition: If $f(n)$ is a polynomial of degree d, that is, $f(n) = a_0 + a_1 n + \cdots + a_d n^d$, and $a_d > 0$, then $f(n)$ is $O(n^d)$.
- Justification:
- Note that, for $n \geq 1$, we have $1 \leq n \leq n^2 \leq \cdots \leq n^d$; hence, $a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d \leq (|a_0| + |a_1| + |a_2| + \cdots + |a_d|)n^d$. We show that $f(n)$ is $O(n^d)$ by defining $c = |a_0| + |a_1| + \cdots + |a_d|$ and $n0 = 1$.

Example#1

- $5n^2 + 3n\log n + 2n + 5$ is $O(n^2)$.

- Justification: $5n^2+3n\log n+2n+5 \leq (5+3+2+5)n^2 = cn^2$, for $c=15$, when $n \geq n0 = 1$.

Example#2

- $20n^3 + 10n\log n + 5$ is $O(n^3)$.
- Justification: $20n^3 + 10n\log n + 5 \le 35n^3$, for $n \ge 1$.

Example#3

- $3\log n+2$ is $O(\log n)$.
- Justification: $3\log n+2 \leq 5\log n$, for $n \geq 2$. Note that $\log n$ is zero for $n = 1$. That is why we use $n \geq n0 = 2$ in this case.

Example#4

- 2^{n+2} is $O(2^n)$.
- Justification: $2^{n+2} = 2^n \cdot 2^2 = 4 \cdot 2^n$; hence, we can take $c = 4$ and $n0 = 1$ in this case.

Example#5

- $2n+100\log n$ is $O(n)$.
- Justification: $2n+100\log n \leq 102n$, for $n \geq n_0 = 1$; hence, we can take $c = 102$ in this case.

Characterizing Functions in Simplest Terms

- While it is true that the function $f(n) = 4n^3 + 3n^2$ is $O(n^5)$ or even $O(n^4)$, it is more accurate to say that $f(n)$ is $O(n^3)$.
- It is not preferable to say that the function $2n^2$ is $O(4n^2 + 6n\log n)$, although this is completely correct.

O, Ω, and ϴ

- **Big O notation (O)**
  - It is defined as upper bound.
  - Upper bound on an algorithm is the most amount of time required (the worst-case performance).
- **Big Omega notation (Ω)**
  - It is defined as lower bound
  - Lower bound on an algorithm is the least amount of time required (the most efficient way possible, in other words best case).
- **Big Theta notation (ϴ):**
  - It is defined as tightest bound and tightest bound is the best of all the worst-case times that the algorithm can take.

Big-Omega

- Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $\Omega(g(n))$, pronounced " $f(n)$ is big-Omega of $g(n)$," if $g(n)$ is $O(f(n))$, that is, there is a real constant $c > 0$ and an integer constant $n0 \geq 1$ such that $f(n) \geq cg(n)$, for $n \geq n0$.
- Example: 3nlog n−2n is Ω(nlog n).
  - Justification: 3nlog n−2n = nlog n+2n(logn−1) ≥ nlog n for n ≥ 2; hence, we can take c = 1 and n0 = 2 in this case.

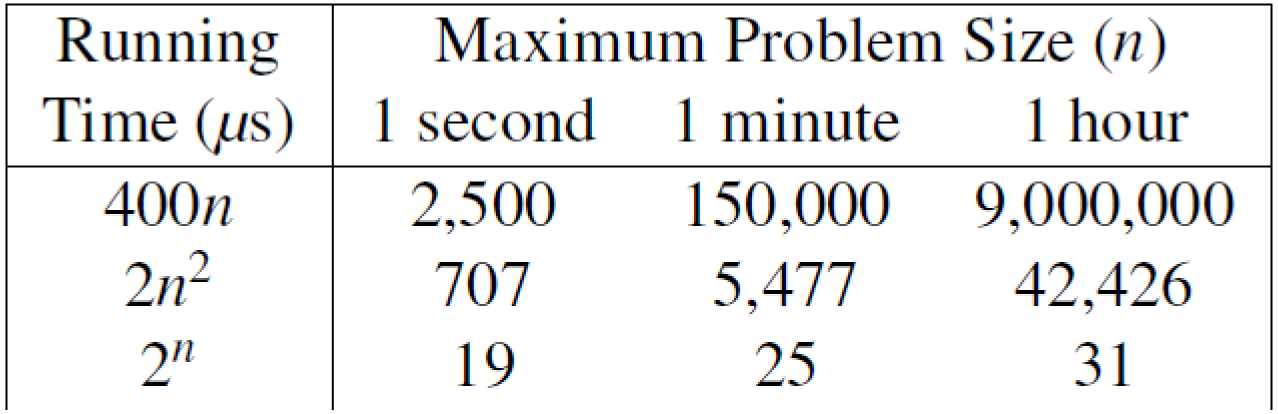
Big-Theta

- This notation allows us to say that two functions grow at the same rate, up to constant factors.
- We say that $f(n)$ is $\Theta(g(n))$, pronounced "$f(n)$ is big-Theta of $g(n)$," if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$, that is, there are real constants $c' > 0$ and $c'' > 0$, and an integer constant $n0 \geq 1$ such that $c'g(n) \leq f(n) \leq c''g(n)$, for $n \geq n0$.

- Example: $3n\log n+4n+5\log n$ is $\Theta(n\log n)$.
  - Justification: $3n\log n \leq 3n\log n+4n+5\log n \leq (3+4+5)n\log n$ for $n \geq 2$.

Comparative Analysis

- Suppose two algorithms solving the same problem are available: an algorithm A, which has a running time of $O(n)$, and an algorithm B, which has a running time of $O(n^2)$. Which algorithm is better? We know that n is $O(n^2)$, which implies that algorithm A is ***asymptotically better*** than algorithm B, although for a small value of n, B may have a lower running time than A.
- Maximum size of a problem that can be solved in 1 second, 1 minute, and 1 hour, for various running times measured in microseconds.



Running	Maximum Problem Size (n)		
Time (μs)	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

Comparative Analysis (Cont.)

- The importance of good algorithm design goes beyond just what can be solved effectively on a given computer. If we achieve a dramatic speedup in hardware, we still cannot overcome the slow algorithm.
- For example, while it is true that the function $10^{100}n$ is $O(n)$, if this is the running time of an algorithm being compared to one whose running time is $10n\log n$, we should prefer the $O(n\log n)$, even though the linear-time algorithm is asymptotically faster. This preference is because the constant factor, 10^{100}

Comparative Analysis (Cont.)

- The observation above raises the issue of what constitutes a “fast” algorithm.
- Generally speaking, any algorithm running in $O(n \log n)$ time should be considered efficient. Even an $O(n^2)$-time function may be fast enough in some contexts, that is, when n is small.
- But an algorithm running in $O(2^n)$ time should almost never be considered efficient.

Constant-Time Operations

- A call to the function, len(data), is evaluated in constant time, where data is a Python list.
  - Because the list class maintains, for each list, an instance variable that records the current length of the list.
  - We say that this function runs in $O(1)$ time.

- Using syntax, data[j], for integer index j. Because Python's lists are implemented as ***array-based sequences***, references to a list's elements are stored in a consecutive block of memory. Therefore, we say that the expression data[j] is evaluated in $O(1)$ time for a Python list.

Finding the Maximum of a Sequence

```
def find max(data):
    biggest = data[0] # The initial value to beat
    for val in data:            # For each value:
        if val > biggest  # if it is greater than the best so far,
            biggest = val # we have found a new best (so far)
    return biggest              # When loop ends, biggest is the max
```

- Finding max function runs in $O(n)$ time.

Prefix Averages

- Given a sequence S consisting of n numbers, we want to compute a sequence A such that $A[j]$ is the average of elements $S[0], \ldots, S[j]$, for $j = 0, \ldots, n-1$, that is,

$$A[j] = \frac{\sum_{i=0}^{j} S[i]}{j+1}$$

A Quadratic-Time Algorithm

```
def prefix_average1(S):
    n = len(S)
    A = [0]*n                     # create new list of n zeros
    for j in range(n):
        total = 0                 # begin computing S[0] + ... + S[j]
        for i in range(j + 1):
            total += S[i]
        A[j] = total / (j+1)      # record the average
    return A
```

A Quadratic-Time Algorithm (Cont.)

- The statement `total += S[i]`, in the inner loop, is executed $1+2+3+\cdots+n$ times.
- $1+2+3+\cdots+n = n(n+1)/2$, which implies that the statement in the inner loop contributes $O(n^2)$ time.
- The running time of `prefix average1` is $O(n^2)$.

A Quadratic-Time Algorithm (Cont.)

```
def prefix average2(S):
    n = len(S)
    A = [0]*n              # create new list of n zeros
    for j in range(n):
      A[j] = sum(S[0:j+1]) / (j+1) # record the average
    return A
```

- Even though the expression, sum(S[0:j+1]), seems like a single command, it is a function call and an evaluation of that function takes $O(j+1)$ time in this context. So the running time of prefix average2 is still dominated by a series of steps that take time proportional to $1+2+3+\cdots+n$, and thus $O(n^2)$.

A Linear-Time Algorithm

```
def prefix average3(S):
    n = len(S)
    A = [0]*n                # create new list of n zeros
    total = 0
    for j in range(n):
        total += S[j]
        A[j] = total / (j+1)
    return A
```

- The running time of prefix average3 is $O(n)$, which is much better than the quadratic time of algorithms prefix average1 and prefix average2.

Three-Way Set Disjointness

- Return True if there is no element common to all three lists

```
def disjoint1(A, B, C):
    for a in A:
        for b in B:
            for c in C:
                if a == b == c:
                    return False                # we found a common value
    return True                                 # if we reach this, sets are disjoint
```

- If each of the original sets has size n, then the worst-case running time of this function is $O(n^3)$.

Three-Way Set Disjointness (Cont.)

```
def disjoint2(A, B, C):
    for a in A:
        for b in B:
            if a == b:
                for c in C:
                    if a == c
                        return False # we found a common value
    return True # if we reach this, sets are disjoint
```

- The *worst-case* running time for disjoint2 is $O(n^2)$. There are quadratically many pairs (a,b) to consider. However, if ***A* and *B* are each sets of distinct elements**, there can be at most $O(n)$ such pairs with a equal to b. Therefore, the innermost loop, over C, executes at most n times.

NP-Complete Complexity

- **Decision problems** – A decision problem has only two possible outputs (yes or no) on any input.
- **NP-complete** problems are a subset of the larger class of **NP (nondeterministic polynomial time) problems**.
- A nondeterministic algorithm is one that, given the same input, can produce different outputs or follow different paths through its execution on different runs.
- **NP** problems are a class of computational problems that can be solved in polynomial time by a non-deterministic machine and can be verified in polynomial time by a deterministic Machine.
- A decision problem **L** is **NP-complete** if it follow the below two properties:

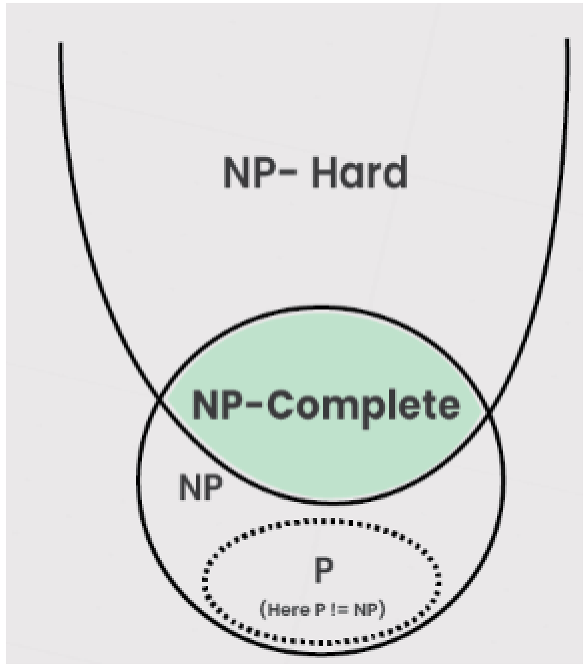
  1. **L** is in **NP** (Any solution to NP-complete problems can be checked quickly, but no efficient solution is known).

  2. Every problem in **NP** is reducible to **L** in polynomial time.

NP-Complete Complexity (Cont.)

- A problem is **NP-Hard** if it obeys Property 2 and need not obey Property 1. Therefore, a problem is **NP-complete** if it is both **NP** and **NP-hard**.

- If any **NP-complete** problem can be solved in polynomial time, then every problem in **NP** can be solved in polynomial time.



SAT is NP Complete

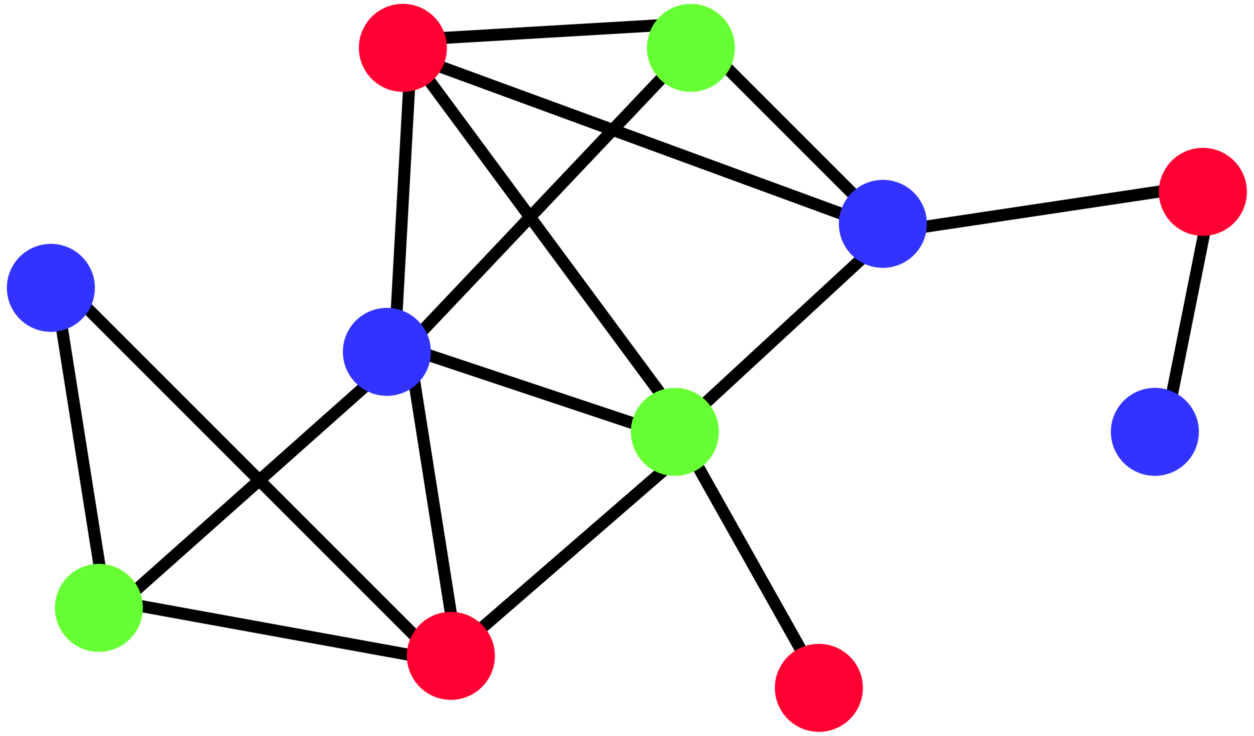
- SAT(Boolean Satisfiability Problem) is the problem of determining if there exists an interpretation that satisfies a given boolean formula.
- It asks whether the variables of a given boolean formula can be consistently replaced by the values **TRUE or FALSE** in such a way that the formula evaluates to **TRUE**. If this is the case, the formula is called *satisfiable*.
- On the other hand, if no such assignment exists, the function expressed by the formula is **FALSE** for all possible variable assignments and the formula is *unsatisfiable*.

Set partition is NP complete

- **Set partition problem:** Set partition problem partitions an array of numbers into two subsets such that the sum of each of these two subsets is the same.
- **Problem Statement:** Given a set **S** of **N** numbers, the task is to determine if the set contains two partitions of **S**, with both of them having exactly the same sum.

3-coloring is NP Complete

- **Problem Statement:** Given a graph **G(V, E)** and an integer K = 3, the task is to determine if the graph can be colored using **at most 3** colors such that no two adjacent vertices are given the same color.



Undecidable Problems

- **No algorithmic solution exists**
    - **Regardless of cost**
    - **These problems aren’t computable**
    - **No answer can be obtained in finite amount of time**

The Halting Problem

- **Definition:** The Halting Problem asks whether a given program or algorithm will eventually halt (terminate) or continue running indefinitely for a particular input. “Halting” means that the program will either accept or reject the input and then terminate, rather than going into an infinite loop.

- Can we create an algorithm that determines whether any given program will halt for a specific input?

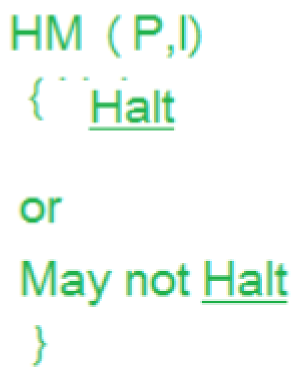
- **Answer:** No, it is impossible to design a generalized algorithm that can accurately determine whether any arbitrary program will halt. The only way to know if a specific program halts is to run it and observe the outcome. This makes the Halting Problem an **undecidable problem**.

The Halting Problem: Proof by Contradiction

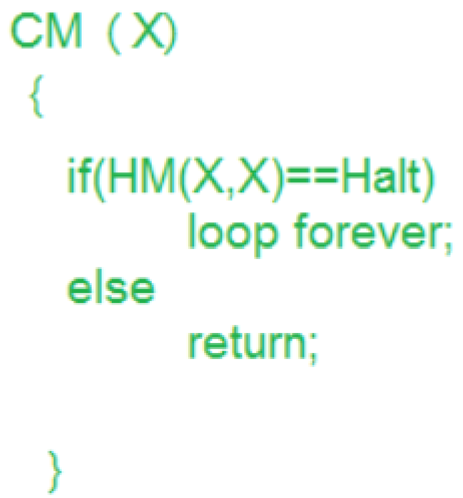
- **Assumption:** Suppose we can design such a machine, called **HM(P, I)**, where:
  - **HM** is the hypothetical machine/program.
  - **P** is the program.
  - **I** is the input.
  - When provided with these inputs, HM will determine whether program P halts or not.

The Halting Problem: Proof by Contradiction (Cont.)

- **Constructing a Contradiction:** Using HM, we can create another program, called **CM(X)**, where **X** is any program passed as an argument as shown in figure.



```
HM (P,I)
 { Halt

 or

 May not Halt
 }
```



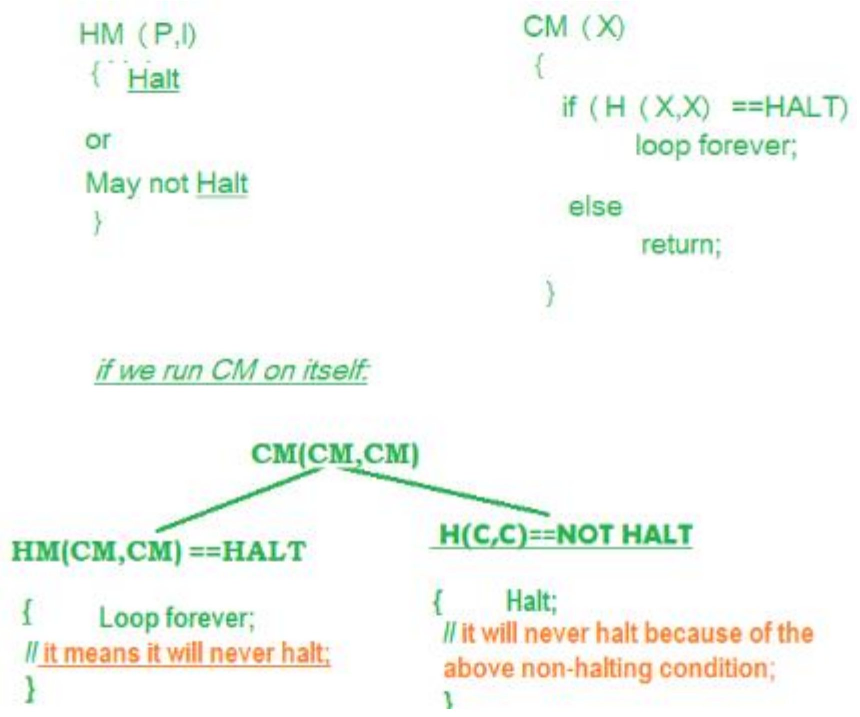
```
CM (X)
 {
   if(HM(X,X)==Halt)
          loop forever;
   else
          return;

  }
```

The Halting Problem: Proof by Contradiction (Cont.)

- **Behavior of CM(X):** In CM(X), we call HM with inputs (X, X), meaning we pass program X both as the program and as its input. HM(X, X) will return either “Halt” or “Not Halt.”
    - If HM(X, X) predicts that X halts, CM(X) will enter an infinite loop.
    - If HM(X, X) predicts that X does not halt, CM(X) will halt immediately.

The Halting Problem: Proof by Contradiction (Cont.)

- **The Contradiction:** Now, consider what happens if we pass CM itself as an argument: **CM(CM)**.
- If HM predicts that CM(CM) halts, CM will loop indefinitely (contradicting HM's prediction).
- If HM predicts that CM(CM) does not halt, CM will halt immediately (again contradicting HM).



```
HM (P,I)
 { Halt

 or
 May not Halt
 }
```

```
CM (X)
{
   if (H (X,X) ==HALT)
          loop forever;
   else
          return;
}
```

if we run CM on itself:

```
{      Loop forever;
// it means it will never halt;
}
```

```
{      Halt;
// it will never halt because of the
above non-halting condition;
}
```