Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

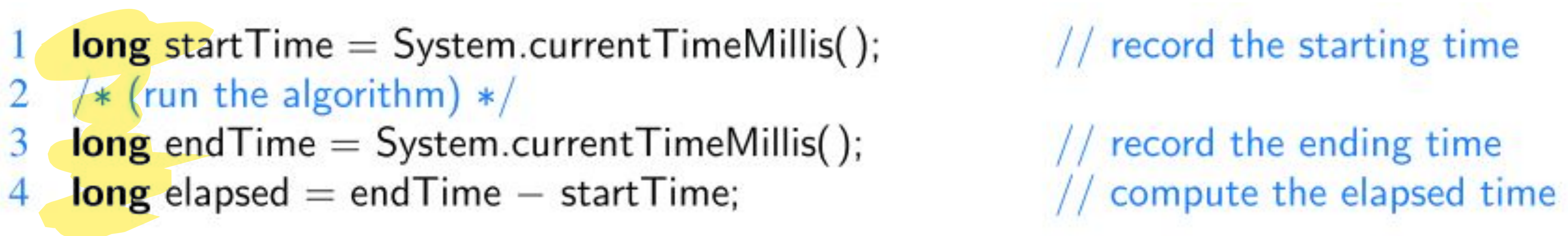
Slides are prepared from

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

Analysis of Algorithm

- One way to study the efficiency of an algorithm is to implement it and experiment by running the program on various test inputs while recording the time spent during each execution.
- A simple mechanism for collecting such running times in Java is based on use of the currentTimeMillis method of the System class.
- That method reports the number of milliseconds that have passed since a benchmark time known as the epoch (January 1, 1970 UTC).
- If we record the time immediately before executing the algorithm and then immediately after, we can measure the elapsed time of an algorithm's execution by computing the difference of those times.



```java
long startTime = System.currentTimeMillis();        // record the starting time
/* (run the algorithm) */
long endTime = System.currentTimeMillis();          // record the ending time
long elapsed = endTime − startTime;                 // compute the elapsed time
```

Code Fragment 4.1: Typical approach for timing an algorithm in Java.

Which computer is best for test?

- The measured times reported by both methods *currentTimeMillis* and *nanoTime* will vary greatly from machine to machine, and may likely vary from trial to trial, even on the same machine.
- Many processes share use of a computer's central processing unit (or CPU) and memory system; therefore, the elapsed time will depend on what other processes are running on the computer when a test is performed
- Experiment Challenges
    - 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; hence, they leave out the running times of inputs not included in the experiment
    - An algorithm must be fully implemented in order to execute it to study its running time experimentally.

Moving Beyond Experimental Analysis (Theoretical Analysis)

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

How to do such Analysis?

- Step-1: Counting Primitive Operations
  - Perform an analysis directly on a high-level description of the algorithm
  - A primitive operation corresponds to a low-level instruction with an execution time that is constant
  - This operation count will correlate to an actual running time in a specific computer
- Step-2: Measuring Operations as a Function of Input Size
  - 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***.
- Step-3: Focusing on the Worst-Case Input
  - Characterize running times in terms of the worst case, as a function of the input size, ***n***, of the algorithm.
  - An average-case analysis usually requires that we calculate expected running times based on a given input distribution, which usually involves sophisticated probability theory

Mathematical Functions

- The following seven most important functions used in the analysis of algorithms
  1. The Constant Function
  2. The Logarithm Function
  3. The Linear Function
  4. The N-Log-N Function
  5. The Quadratic Function
  6. The Cubic Function and Other Polynomials
  7. The Exponential Function

The Constant Function

- The simplest function we can think of is the constant function, that is,

  $$f(n) = c,$$

  for some fixed constant ***c***, such as c = 5, c = 27, or c = 2^{10}
- For any argument ***n***, the constant function ***f(n)*** assigns the value ***c***
- We are most interested in integer functions that will give answer in integer value
  - the most fundamental constant function is **g(n) = 1**
- Any other constant function, ***f(n) = c***, can be written as a constant ***c*** times ***g(n)***. That is, ***f(n) = c * g(n)*** in this case
- The constant function characterizes the number of steps needed to do a basic operation on a computer, like
  - adding two numbers,
  - assigning a value to a variable, or
  - comparing two numbers

The Logarithm Function

$$f(n) = \log_b n$$

- for some constant ***b > 1***
- This function is defined as the inverse of a power, as follows:

$$x = \log_b n \text{ if and only if } (b)^x = n$$

- The value ***b*** is known as the base of the logarithm. Note that by the above definition, for any base ***b > 0***, we have that $\log_b 1 = 0$.
- We use log function because computers store integers in binary format
- The ceiling of ***x*** can be viewed as an integer approximation of ***x*** since we have

$$x \leq \lceil x \rceil < x+1$$

- Logarithm Rules: Given real numbers a > 0, b > 1, c > 0, and d > 1, we have:

1. $\log_b(ac) = \log_b a + \log_b c$
2. $\log_b(a/c) = \log_b a - \log_b c$
3. $\log_b(a^c) = c\log_b a$
4. $\log_b a = \log_d a/\log_d b$
5. $b^{\log_d a} = a^{\log_d b}$

The Linear Function

$f(n) = n$

- Given an input value ***n***, the linear function ***f*** assigns the value ***n*** itself

- This function arises in algorithm analysis any time we have to do a single basic operation for each of ***n*** elements
    - For example, comparing a number ***x*** to each element of an array of size ***n*** will require ***n*** comparisons

The N-Log-N Function

f(n) = nlogn

- The function that assigns to an input ***n*** the value of ***n*** times the logarithm base-two of ***n***
- This function grows a little more rapidly than the linear function and a lot less rapidly than the quadratic function
- Several important algorithms that exhibit a running time proportional to the ***n-log-n*** function

The Quadratic Function

f(n) = n^2

- Given an input value ***n***, the function ***f*** assigns the product of ***n*** with itself
- The main reason why the quadratic function appears in the analysis of algorithms is that there are many algorithms that have nested loops
- 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

$$1+2+3+\cdots+(n-2)+(n-1)+n = \frac{n(n+1)}{2}.$$

- The total number of operations that will be performed by the nested loop if the number of operations performed inside the loop increases by one with each iteration of the outer loop

The Cubic Function and Other Polynomials

f(n) = n³

- Assigns to an input value ***n*** the product of ***n*** with itself three times
- The cubic function appears less frequently in the context of algorithm analysis than the constant, linear, and quadratic functions
- **Polynomials:**The linear, quadratic and cubic functions can each be viewed as being part of a larger class of functions, the polynomials

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d,$$

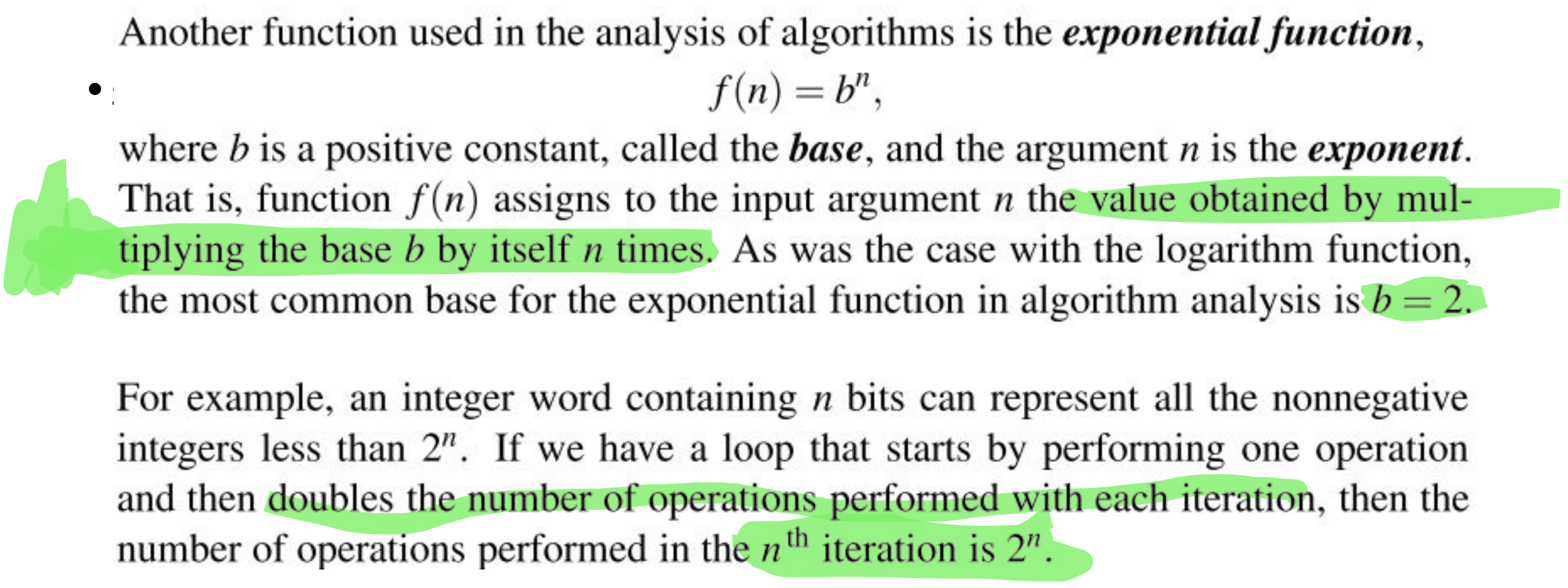
where $a_0, a_1, \ldots, a_d$ are constants, called the ***coefficients*** of the polynomial, and $a_d \neq 0$. Integer d, which indicates the highest power in the polynomial, is called the ***degree*** of the polynomial.

- **Summation**

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b),$$

where a and b are integers and $a \leq b$. Summations arise in data structure and algorithm analysis because the running times of loops naturally give rise to summations.

The Exponential Function



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

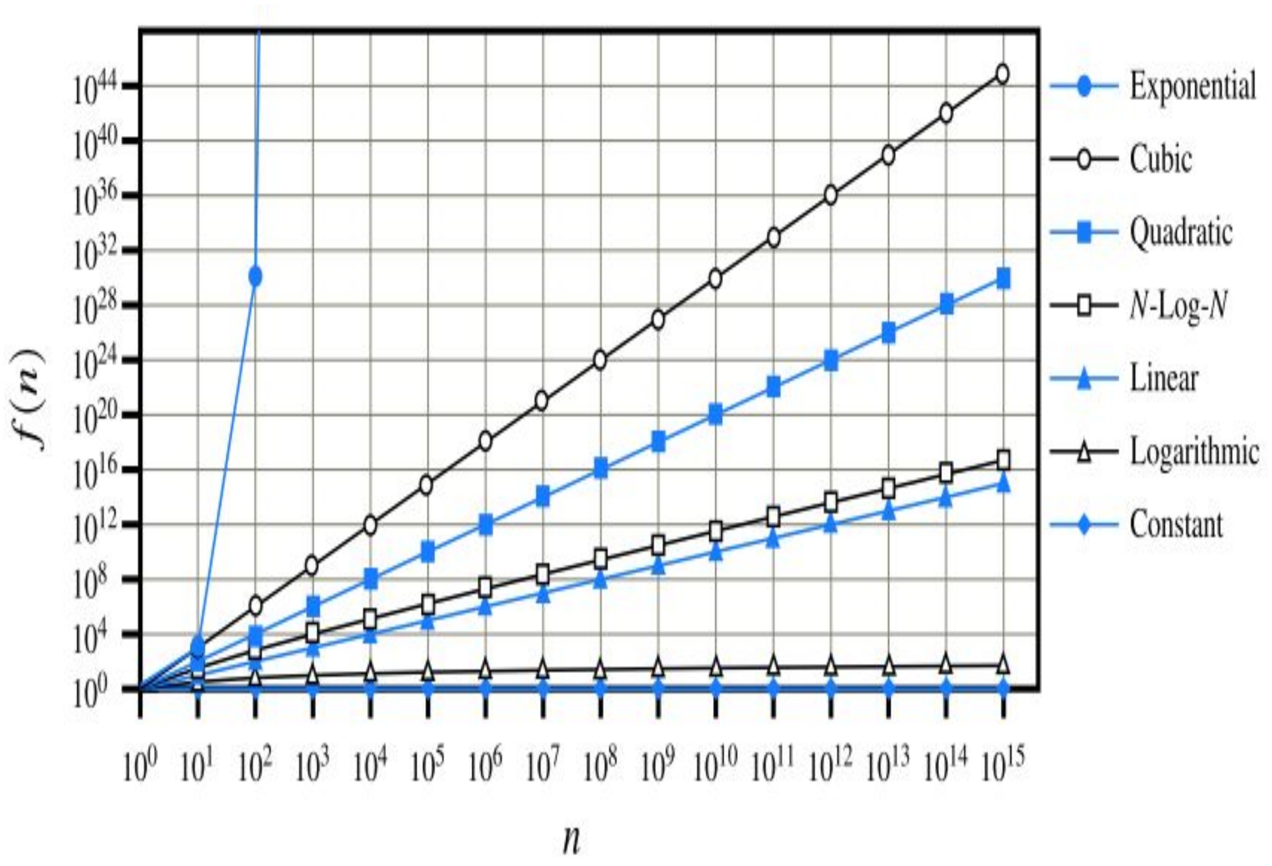
$$f(n) = b^n,$$

where b is a positive constant, called the ***base***, and the argument n is the ***exponent***. That is, function $f(n)$ assigns to the input argument n the value obtained by multiplying the base b by itself n times. As was the case with the logarithm function, the most common base for the exponential function in algorithm analysis is $b = 2$.

For example, an integer word containing n bits can represent all the nonnegative integers less than 2^n. If we have a loop that starts by performing one operation and then doubles the number of operations performed with each iteration, then the number of operations performed in the nth iteration is 2^n.

Comparing Growth Rates

- Ideally, we would like data structure operations to run in times proportionalto the constant or logarithm function
- we would like our algorithms to run in linear or n-log-n time.
- Algorithms with quadratic or cubic running times are less practical
- Algorithms with exponential running times are infeasible for all but the smallest sized inputs.


f(n)
10^44
10^40
10^36
10^32
10^28
10^24
10^20
10^16
10^12
10^8
10^4
10^0
10^0 10^1 10^2 10^3 10^4 10^5 10^6 10^7 10^8 10^9 10^10 10^11 10^12 10^13 10^14 10^15
n
Exponential
Cubic
Quadratic
N-Log-N
Linear
Logarithmic
Constant