

Chapter 2

Introduction to Algorithm Analysis

The Main Topics

- Learn how to investigate the efficiency of an algorithm
- Learn about the order of growth
- Learn about asymptotic notations

Introduction to Algorithm Analysis

- The speed and the memory of today's computers are two primary important considerations from a practical point of view
- “Analysis of algorithms” means an investigation of an algorithm's efficiency with respect to two resources:
 - running time
 - memory space

Introduction to Algorithm Analysis

- There are two kinds of efficiency:
 - *Time efficiency* indicates how fast an algorithm runs
 - *Space efficiency* refers to the amount of memory required by an algorithm
- In this chapter, we primarily concentrate on time efficiency

Measuring an Input's Size


- Almost all algorithms run longer on larger inputs
 - It is logical to investigate an algorithm's efficiency as a function of some parameter n indicating the algorithm's input size: $f(n)$
 - It is understood that the value of n must be large enough
- ➡ How large is large enough?

Examples

Measuring an Input's Size

- Almost all algorithms run longer on larger inputs
- It is logical to investigate an algorithm's efficiency as a function of some parameter n indicating the algorithm's input size: $f(n)$
 - It is understood that the value of n must be large enough
 - ➡ How large is large enough?
- In most cases, selecting such a parameter is quite straightforward

Units for Measuring Running Time

- Some standard unit of time measurement (second, millisecond, ...) may be used to measure the running time of a program implementing the algorithm
 - There are some drawbacks to such an approach:
 - the language used in writing the program
 - programmer's skill
 - the speed of a particular computer
 - the compiler used in generating the machine code
 - ...
 - the test set
-  A metric that does not depend on these factors is needed

Units for Measuring Running Time

- One possible approach is to count the number of times each of the algorithm's operations is executed
 - ➡ This approach is both excessively difficult and usually unnecessary
 - The solution is to ...
 - (i) identify the operation contributing the most to the total running time, and
 - (ii) compute the number of times this operation is executed
- ➡ Such an operation is called the *basic operation*

The Basic Operation

- It is usually the most time-consuming operation
- It is usually in the algorithm's innermost loop
- It usually relates to the data that needs to be processed



The time efficiency of an algorithm is measured by counting the number of times the algorithm's basic operation is executed on inputs of size n

Order of Growth of the Running Time

- Assume that there are two different algorithms which are designed to solve the same problem
 - Let $f_1(n) = 0.1n^2 + 10n + 100$ and $f_2(n) = 0.1n^2$ be the number of times the basic operation is executed by the first and second algorithm, respectively

| n | $f_1(n)$ | $f_2(n)$ | $f_1(n)/f_2(n)$ |
|--------|--------------|---------------|-----------------|
| 10^1 | 210 | 10 | 21 |
| 10^2 | 2100 | 1000 | 2.1 |
| 10^3 | 110100 | 100000 | 1.101 |
| 10^6 | 100010000100 | 1000000000000 | 1.000100001 |

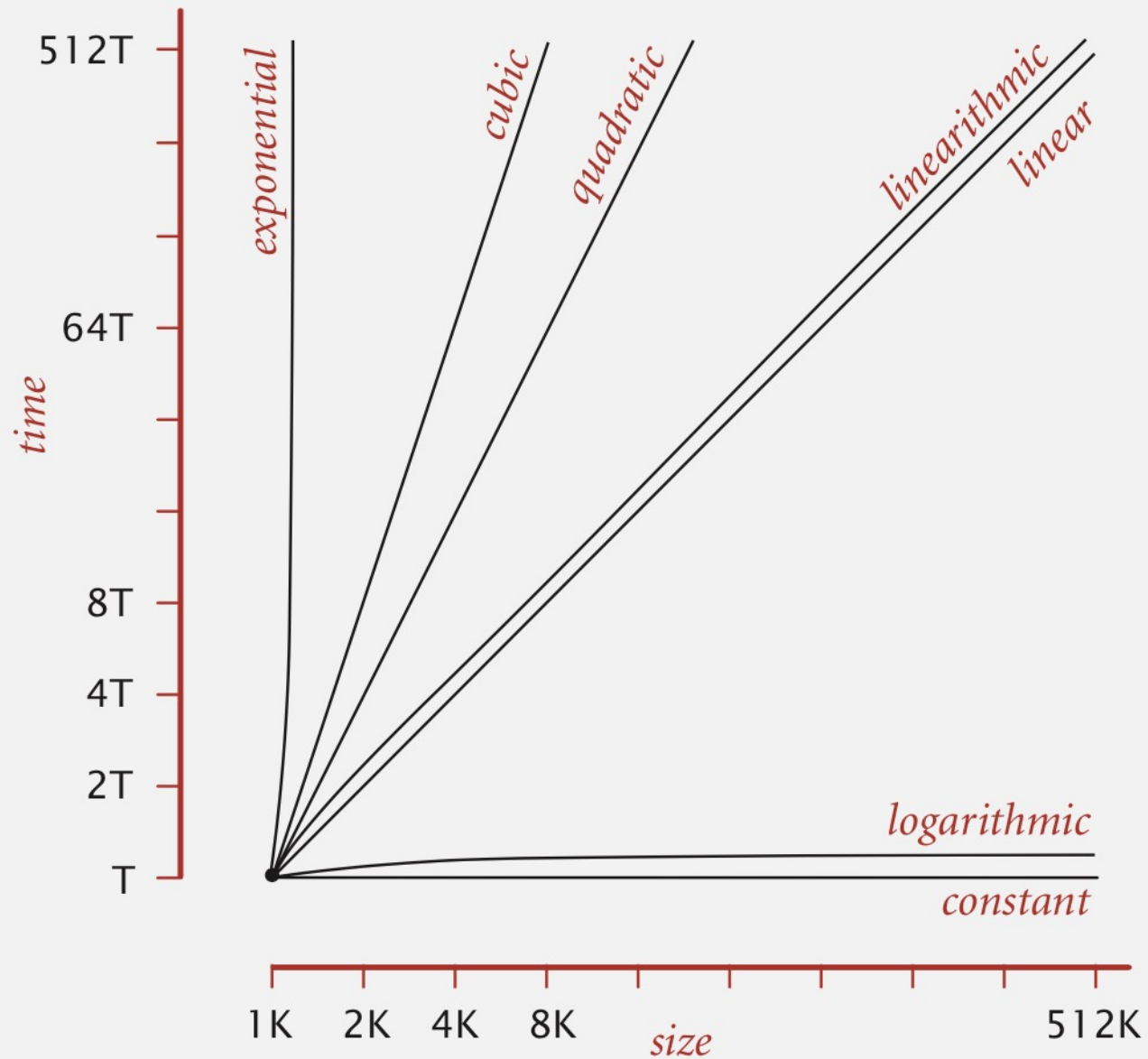
Order of Growth of the Running Time

- Since the lower-order terms are relatively insignificant for large values of n , *the leading term of a polynomial is considered only*
- ➡ It's called the *order of growth* of the running time
- The leading term's constant coefficient is also ignored
 - Since it is less significant than the order of growth in determining computational efficiency for large inputs

The Typical Growth Rates

| Function | Name |
|-------------------|-------------------------------------|
| $O(1), \Theta(1)$ | Constant time |
| $\log n$ | Logarithmic time |
| n | Linear time |
| $n \log n$ | Linearithmic or Quasilinear time |
| n^k | Polynomial time |
| k^n | Exponential time |
| $n!$ | Factorial time |

log-log plot



Typical orders of growth

Worst-Case, Best-Case, and Average-Case

- The efficiency of many algorithms depends not only on the input size but also on the distribution of a particular input
- Thus, the efficiency of such algorithms needs to be investigated in three cases
 - The *worst-case* running time
 - The *best-case* running time
 - The *average-case* running time

The Worst-Case Running Time

- It indicates the *longest* running time for any input of size n
 - It's an upper bound on the running time for any input
- Most of the times, we do the worst-case analysis to analyze algorithms
- Knowing the worst-case running time provides a guarantee that the algorithm will never take any longer

The Best-Case Running Time

- It indicates the *fastest* running time for any input of size n
 - It's a lower bound on the running time for any input
- The best-case analysis is almost impractical
- Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst-case
 - An algorithm may take years to run

The Average-Case Running Time

- It is an estimate of the average running time for all possible inputs of size n
- The average-case analysis is not easy to do in most of the practical cases and it is rarely done
 - It is considerably more difficult than the investigation of the worst-case and best-case efficiencies
 - It cannot be obtained by taking the average of the worst-case and the best-case efficiencies
- In the average-case analysis, we must know (or predict) the mathematical distribution of all possible inputs

Example

- Consider the sequential search algorithm

```
bool seqSearch(a[1 .. n], k) {  
    i = 1;  
    while (i ≤ n) && (a[i] != k)  
        i++;  
    return i ≤ n;  
}
```

- In the best case: $\Theta(1)$
- In the worst case: $\Theta(n)$
- In the average case?

Example

Asymptotic Notations

- The order of growth of an algorithm is considered as the principal indicator of the algorithm's efficiency
- To compare and rank such orders of growth, computer scientists use three notations:
 - O (big oh)
 - Ω (big omega)
 - Θ (big theta)

Asymptotic Notations

- In the following discussion, $f(n)$ and $g(n)$ can be any nonnegative functions defined on \mathbb{N}
- If $f(n)$ denotes an algorithm's running time, then $g(n)$ denotes some simple function to compare with
 - For simplicity, let $g(n) = n^2$

Big Oh Notation

- Informally, $O(g(n))$ is the set of all functions with a lower or same order of growth as $g(n)$

$$2n + 5 \in O(n^2), 3 \log n + 1 \in O(n^2), \frac{n(n+1)}{2} \in O(n^2)$$

$$5n^3 + 6n^2 \notin O(n^2), 0.00001n^3 \notin O(n^2), 2^n - 1 \notin O(n^2)$$

- A description of a function $f(n)$ in terms of big O notation provides an *upper bound* on the growth rate of the function

Big Omega Notation

- Informally, $\Omega(g(n))$ is the set of all functions with a higher or same order of growth as $g(n)$

$$2n + 5 \notin \Omega(n^2), 3 \log n + 1 \notin \Omega(n^2), \frac{n(n+1)}{2} \in \Omega(n^2)$$

$$5n^3 + 6n^2 \in \Omega(n^2), 0.0001n^3 \in \Omega(n^2), 2^n - 1 \in \Omega(n^2)$$

- A description of a function $f(n)$ in terms of big Ω notation provides a *lower bound* on the growth rate of the function

Big Theta Notation

- Informally, $\Theta(g(n))$ is the set of all functions that have the same order of growth as $g(n)$

$$2n + 5 \notin \Theta(n^2), 3 \log n + 1 \notin \Theta(n^2), \frac{n(n+1)}{2} \in \Theta(n^2)$$

$$5n^3 + 6n^2 \notin \Theta(n^2), 0.0001n^3 \notin \Theta(n^2), 2^n - 1 \notin \Theta(n^2)$$

- Obviously, $\Theta(g(n))$ is a stronger statement than $O(g(n))$ and $\Omega(g(n))$