ENSF 593/594
Data Structures – Analysis of Algorithms

Mohammad Moshirpour

Outline

- Importance of Complexity Analysis
- Complexity
- Asymptotic Complexity
- Upper and Lower Bounds
- Big-O
- Classes of Algorithms
- Ω And Θ Notations
- Best, Worst, and Average-Case Complexities
- Examples

Goal

- In this lecture we will study about the importance of algorithms complexity and how to analyze complexity of algorithms.

Introduction

- We seek algorithms that are:
  - Correct
    - Must be shown to work for all possible inputs
      - Ideally, we provide a formal mathematical *proof*
  - Efficient
    - We prefer algorithms that minimize running time and memory usage, especially for large inputs
    - Is measured by doing *complexity analysis*
      - Allows us to compare competing algorithms
  - Easy to implement

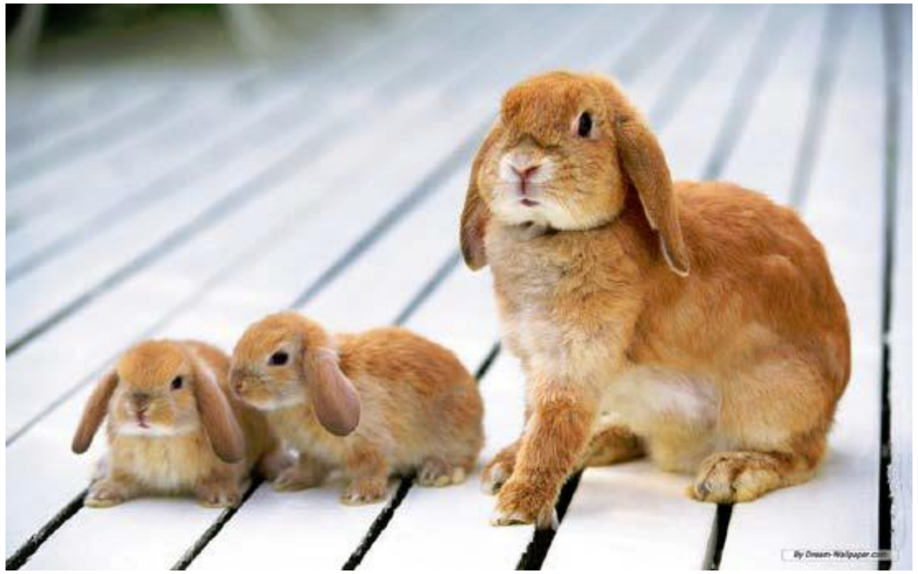
Why Complexity Analysis is Important?

- Just consider the Fibonacci series

1,1,2,3,5,8,13,21,34,55,89,144,…

$$F_n = F_{n-1} + F_{n-2}$$

$$F_1 = 1\,, F_2 = 1$$



Fibonacci Implementations

Iterative

```
int fib(int n)
{
int i = 1, j = 0, k, t;

for (k = 1; k <= n; k++){
t = i + j;
i = j;
j = t;
}
return j;
}
```

Recursive

```
int fib(int n)
{
        if (n < 2)
                return n;
        else
                return fib(n - 1) + fib(n - 2);
 }
```

Fibonacci Implementations

Iterative

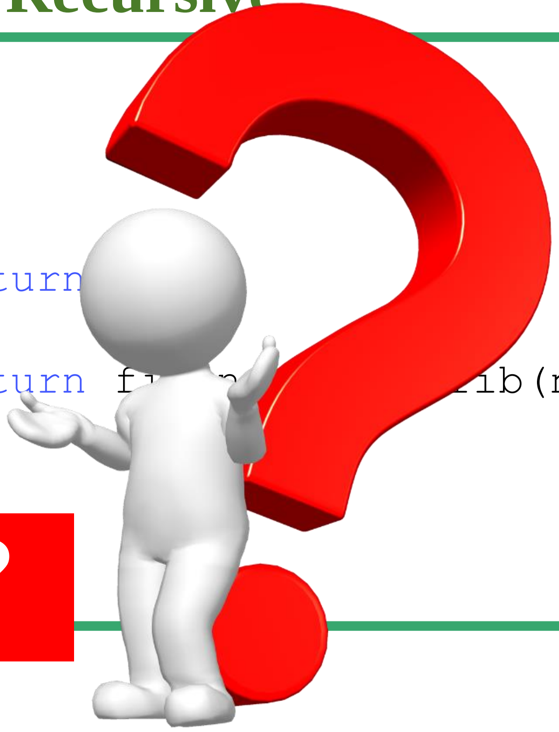
```
int fib(int n)
{
int i = 1, j = 0, k, t;

for (k = 1; k <= n; k++){
t = i + j;
i = j;
j = t;
}
return j;
}
```

Recursive

```
int fib(int n)
{
        if (n < 2)
                return
        else
                return fi   n          ib(n - 2);
 }
```

Which one is better?



Run time of Fibonacci Implementations

- Let's Compute the 1000th Fibonacci number using a Core i7 Quad Core CPU[1] (One of the best currently available processing unit):

Iterative

Recursive

1- 82,300 MIPS at 2.66 (Turbo 2.93) GHz

Run time of Fibonacci Implementations

- Let's Compute the 1000th Fibonacci number using a Core i7 Quad Core CPU[1] (One of the best currently available processing unit):

Iterative

Computes Fib(1000)
After about 7.2×10^{-7} seconds

Recursive

Computes Fib(1000)
after about 4.12×10^{282} years!
(10^{279} times more than age of the Earth!!!)

1- 82,300 MIPS at 2.66 (Turbo 2.93) GHz

Complexity

- Is a measure of the difficulty of performing a computation in terms of:
  - The time required *(time complexity)*, or
  - The number of steps or arithmetic operations required *(computational complexity)*, or
  - The amount of memory required *(space complexity)*
- *Complexity analysis* of an algorithm reveals how the time or space it requires to solve a problem varies with input data size
  - Is expressed as a function of *n* (number of inputs)

Complexity (cont'd)

- We could analyze complexity empirically
  - - i.e. Code and run the algorithm, measuring the time and memory used
  - The results are affected by:
    - Processor speed
    - Ram and disk access time
    - Code produced by the compiler
  - May obscure the true complexity measure

Complexity (cont’d)

- Other limitations
  - Can’t test all possible inputs
  - Can’t compare results unless experiments are done in the same environment
  - Must fully implement the algorithm to test

Complexity (cont’d)

- We prefer to analyze the algorithm directly, ignoring its implementation
  - Count the number of *primitive operations* executed
  - Assume each takes a similar amount of constant time
    - Thus correlates to actual running time on a specific computer
  - Result is some function *t(n)*

Complexity (cont'd)

- E.g. Algorithm: arrayMax(A,*n*)

```
currentMax ← A[0]                       2 ops
i ← 1                                   1 op
while i ≤ n-1 do                        2 ops x n
    if currentMax < A[i] then           2 ops x (n-1)
        currentMax ← A[i]               2 ops x (n-1)
 i ← i + 1                              2 ops x (n-1)
rerurn currentMax                       1 op
```

Complexity (cont'd)

- $t(n)$ is at least: $2 + 1 + 2n + 4(n-1) + 1 = 6n$
  - Best case when A[0] is maximum element
- Or at most: $2 + 1 + 2n + 6(n-1) + 1 = 8n - 2$
  - Worst case when A is in ascending order

Asymptotic Complexity

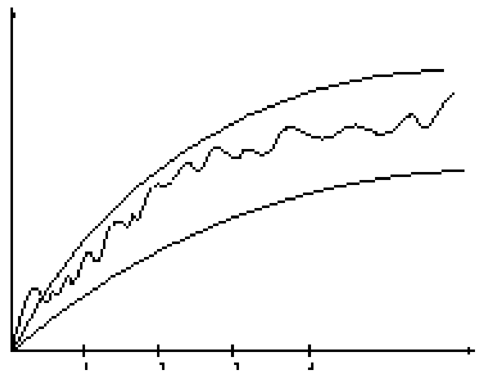
- Measures the *growth rate* of an algorithm as *n* becomes very large
- *Asymptote:* a straight line approached by a curve in the limit as the curve approaches infinity
- As *n* approaches infinity, constants and lower order terms of *t(n)* contribute little to the function’s value
  - Thus are ignored

Asymptotic Complexity (cont'd)

- Allows us to simplify our analysis
  - We ignore details that don't affect our comparisons of algorithms
- E.g. $t(n) = 8n - 2$ grows *linearly* with n
  - i.e. Its true running time is n times a factor that is implementation-dependent

Upper and Lower Bounds

- Allow us to treat complicated functions more simply
  - We ignore small fluctuations, and concentrate on asymptotic growth
- The bound themselves are functions
  - We choose functions that are as simple as possible, yet are reasonably “tight”

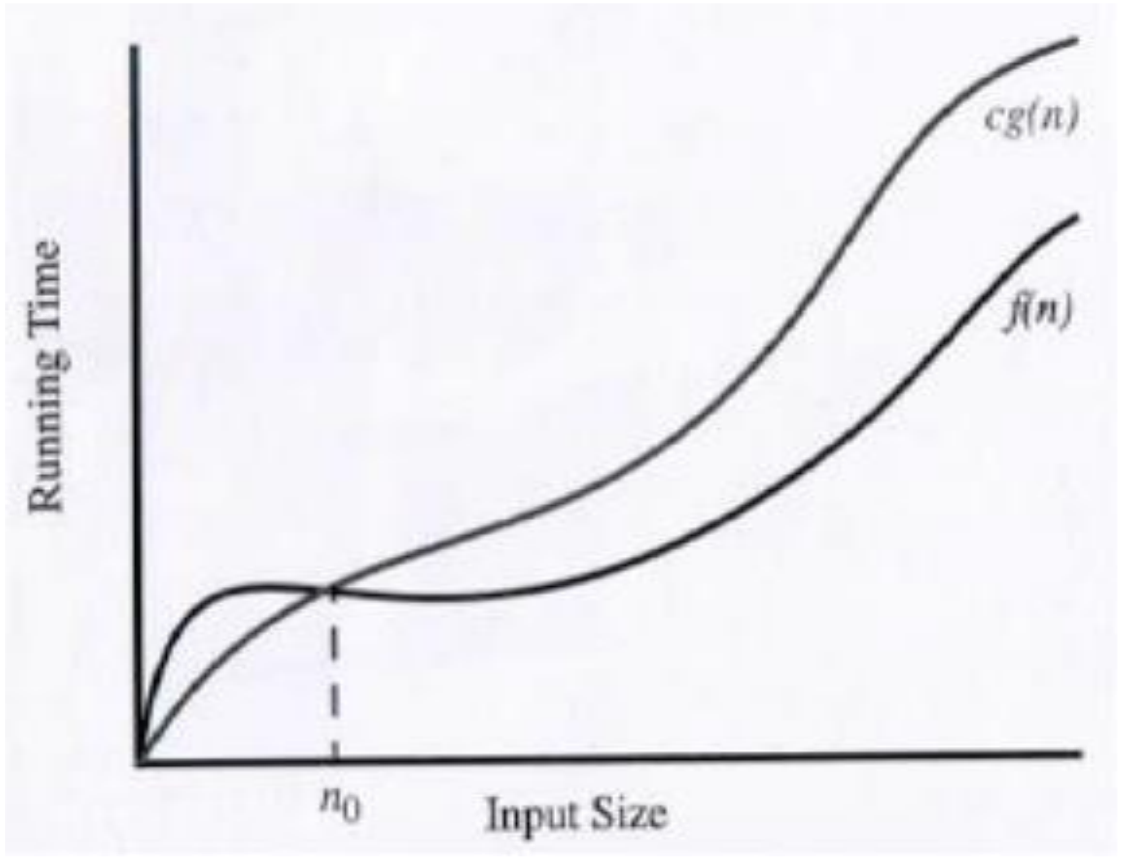


Big-O Notation

- Is used to specify an upper bound on a function
  - $f(n)$ is the function
  - $g(n)$ is an upper bound
- Definition: $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that:

$$f(n) \leq cg(n), \text{ for } n \geq n_0$$

Big-O Notation (cont’d)



Big-O Notation (cont'd)

- E.g. The function $8n - 2$ $O(n)$
  - "is big-Oh of n", or "is order of n"
  - Justification: $8n - 2 \leq cn$ for every integer $n \geq n_0$ when $c = 8$ and $n_0 = 1$
- For a given $g(n)$, there are infinitely many c's and n_0's that can be chosen
  - The point is that $f(n)$ and $g(n)$ grow at the same rate

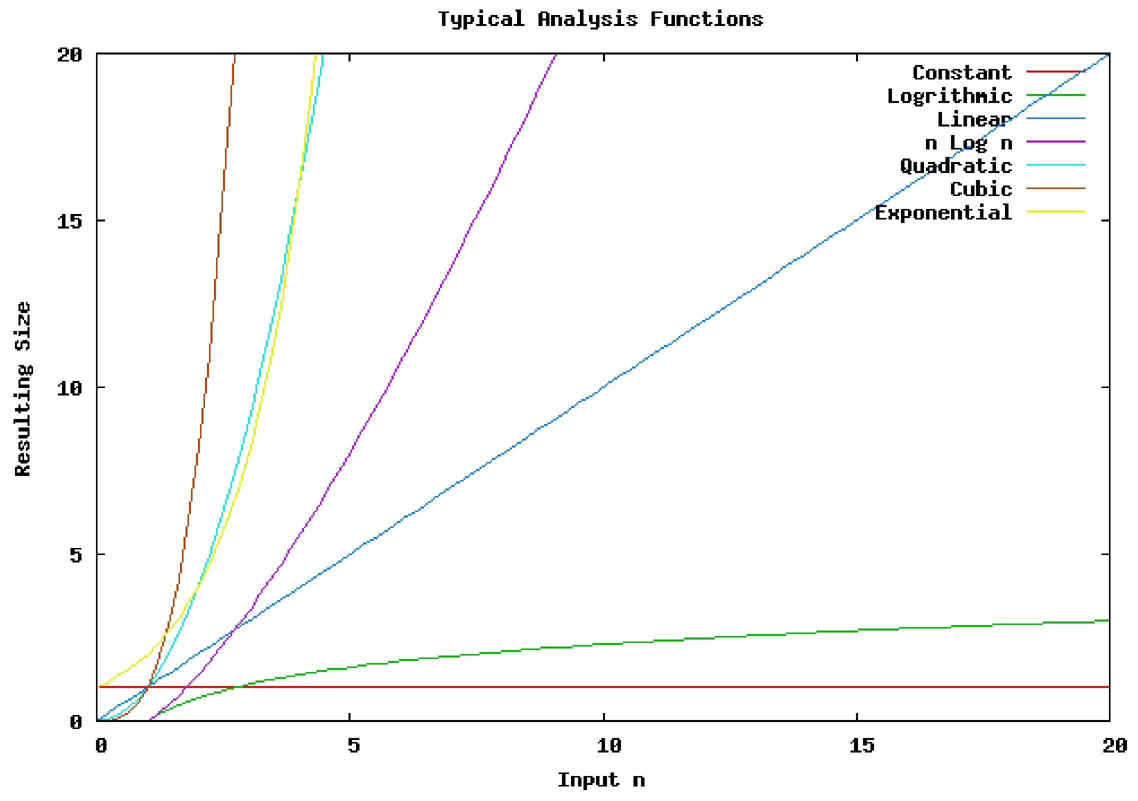
Big-O Notation (cont’d)

- There are infinitely many functions *g* for a given function *f*
  - E.g. $8n - 2$ is also $O(n^2)$, $O(n^3)$, etc.
  - Choose the smallest (simple) function that satisfies the inequality
    - i.e. Choose the tightest bound possible

Classes of Algorithms

- The types of algorithms are:

O(1)	constant
O(lg *n*)	logarithmic
O(*n*)	linear
O(*n* lg *n*)	N-Log-N
O(n^2)	quadratic
O(n^3)	cubic
O(2^n)	exponential



Classes of Algorithms (cont’d)

- N-Log-N algorithms and below are considered efficient, even for large inputs
- Quadratic algorithms and above are practical only with small inputs

Ω And Θ Notations

- Ω ("Big-Omega") notation specifies a lower bound on a function
  - *f(n)* is the function,
  - *g(n)* is a lower bound
- Definition: *f(n)* is Ω(*g(n)*) if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that:

$$f(n) \geq cg(n), \text{ for } n \geq n_0$$

Ω And Θ Notations (Cont'd)

- There are infinitely many functions *g* for the function *f*
  - Choose the largest (simple) function that satisfies the inequality
    - i.e. Choose the tightest bound possible

Ω And Θ Notations (Cont'd)

- Θ ("Big Theta") notation specifies upper and lower bounds simultaneously
- Definition: $f(n)$ is $\Theta(g(n))$ if there are real constants $c_1 > 0$ and $c_2 > 0$ and an integer constant $n_0 \geq 1$ such that:

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ for } n \geq n_0$$

Ω And Θ Notations (Cont'd)

- Θ ("Big Theta") notation specifies upper and lower bounds simultaneously
- Definition: $f(n)$ is Θ($g(n)$) if there are real constants $c_1 > 0$ and $c_2 > 0$ and an integer constant $n_0 \geq 1$ such that:

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ for } n \geq n_0$$

Ω And Θ Notations (Cont’d)

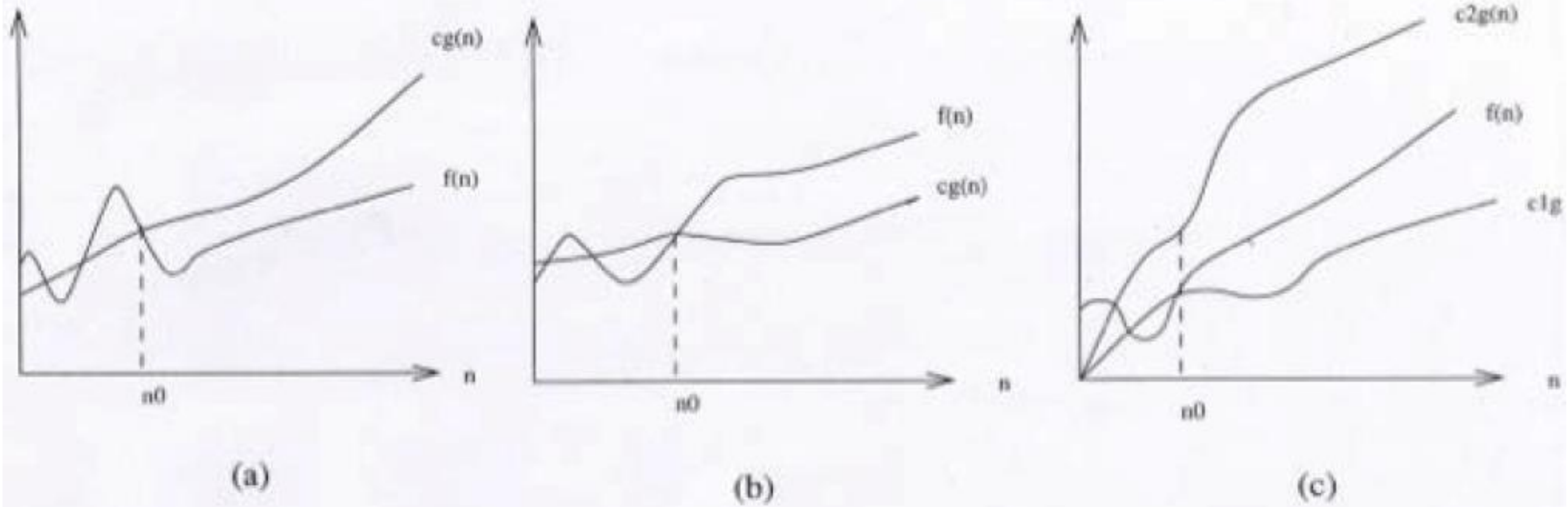
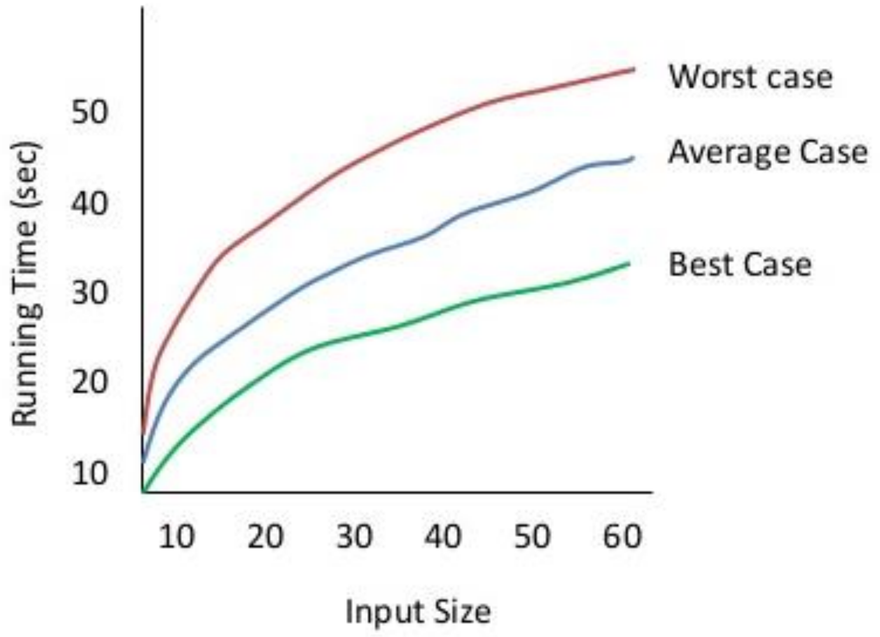


Figure 1–6. Illustrating the (a) O, (b) Ω, and (c) Θ notations

Best, Worst, and Average-Case Complexities (Cont’d)

- Are found by considering all possible arrangement of inputs of size *n*
  - E.g. For sorting, all the possible ordering for a given number of keys
  - Each instance can be plotted graphically

Best, Worst, and Average-Case Complexities (Cont’d)



Best, Worst, and Average-Case Complexities (Cont’d)

- *Worst-case complexity:* is the function defined by the maximum number of steps taken on any input of size *n*
- *Best-case complexity:* is the function defined by the minimum number of steps

Best, Worst, and Average-Case Complexities (Cont'd)

- *Average-case:* is the function defined by the average number of steps
  - We use simple average when each input is equally likely
  - Must use a more complicated analysis for uneven input probability distribution

Rules to Calculate Complexity

- The computational complexity for an algorithm can be found using some basic rules:
  - Simple statements that don’t depend on *0* are *O*(1)
    - i.e. take constant time
  - Ignore differences in execution times for simple statements
    - Multiplicative constants are discarded in big-O analysis
  - Use the worst case for conditional statements
    - i.e. Take the “longest path” through the algorithm
  - If the number of steps is halved on each iteration of a loop, then the complexity is O(lgn)
    - Also true if multiplying by 1/3, ¼, etc.

Rules to Calculate Complexity (Cont'd)

- The computational complexity for an algorithm can be found using some basic rules:
  - Sum rule: if the complexity of a sequence of statements is the sum of two or more terms, discard the lower-order terms
    - E.g. $n^3 + n^2$ is $O(n^3) + O(n^2) = O(n^3)$
  - Product rule: if a process is repeated for each *n* of another process, then *O* is the product of the *O*s of each process
    - E.g. Nested loop processing of a 2-D array is $O(n) \cdot O(n) = O(n \cdot n) = O(n^2)$

Summary

- We seek *algorithms* that are *Correct*, *Efficient* and *Easy to implement*.
- *Complexity analysis* of an algorithm reveals how the time or space it requires to solve a problem varies with input data size.
- *Analyzing Complexity:*

✗ Empirically

✓ Directly (Count the number of primitive operations executed)

Summary

- *Asymptotic Complexity*: as *n* approaches infinity, constants and lower order terms contribute little to the function’s value so they are ignored.
- Upper and lower bounds allow us to treat complicated functions more simply.
- We ignore small fluctuations and concentrate on asymptotic growth.

Summary

- Big-O notation is used to specify an upper bound on a function.
- Common classes of algorithms: constant, logarithmic, linear, N-Log-N, quadratic, cubic, exponential.
- Big-Ω notation specifies a lower bound on a function.
- Big-Θ notation specifies upper and lower bounds simultaneously.

Summary

- There are many functions that satisfies properties of Ω and Θ.
- Best, worst, and average-case complexities can be calculated. Worst-Case is the one that frequently used.

Review Questions

- What is complexity of algorithm?
- What are the two ways to analyze complexity?
- What are the limitation of analyzing complexity empirically?
- What is Asymptotic Complexity and why do we use that?
- Why do we use upper and lower bound for analysis a function?
- What is Big-O?
- What are classes of algorithm in term of Big-O?

Review Questions

- What is Ω notation?
- What is Θ notation?
- There are infinitely many functions as Big-O, Ω and Θ for a given function *f*. Which one should be selected?
- What are best case, average case and worst case of an algorithm?

Any questions?