Data Structures and Algorithms

Lecture 5: Analysis of Algorithms

Department of Computer Science & Technology
United International College

Outline

- Algorithm
  - What is an algorithm?
  - How to describe an algorithm?
- Analysis of Algorithms
- Growth Rate and the Big-Oh Notation

What is an Algorithm?

- A clearly specified set of simple instructions to be followed to solve a problem
  - Takes a set of values, as input and
  - produces a value, or set of values, as output
- Data structures
  - Methods to manipulate data
- Program = algorithms + data structures

An Algorithm May be Described

In English

As pseudo-code

As a program

Example for Algorithm Specification

- Problem: Given a student score, decide whether the student Passes or Fails the course.
- Algorithm:

English

If the student's score is greater or equal to 60, write "Pass".

Otherwise, write "Fail".

Pseudo-Code

```
IF score >= 60
    WRITE "Pass"
ELSE
    WRITE "Fail"
```

C Program

```c
#include <stdio.h>
void judge(int score)
{
    if(score >= 60)
        puts("Pass");
    else
        puts("Fail");
}
```


```
#include <stdlib.h>
Node* InsertNode(Node** phead, int index, double x) {
    if (index < 0) return 0;

    int currIndex    = 1;
    Node* currNode = *phead;
    while (currNode && index > currIndex) {
        currNode = currNode->next;
        currIndex ++;
    }
    if (index > 0 && currNode == 0) return 0;

    Node* newNode    = (Node*)malloc(sizeof(Node));
    newNode->data    = x;
    if (index == 0) {
        newNode->next    = *phead;
        *phead = newNode;
    }
    else {
        newNode->next    = currNode->next;
        currNode->next = newNode;
    }
    return newNode;
}
```

- A combination of human language and programming language
  - Mimics the syntax of a programming language
  - Ignores implementation details
  - A bridge from an idea to a program
- How to Write Pseudocode?

Algorithm Analysis - Why

- Why need algorithm analysis ?
  - writing a working program is not good enough
  - The program may be inefficient!
  - If the program is run on a large data set, then the running time becomes an issue

Example: One Of

- Problem:
  Given an array *A* of *n* sorted values, check whether a value *x* is one of them.
- Algorithm 1 (linear search):

```
FOR EACH value IN A
  IF value = x
     RETURN True
RETURN False
```

Example: One Of

- Algorithm 2:

```
FOR EACH value IN A
    IF value = x
        RETURN True
    ELSE IF value > x
        RETURN False
RETURN False
```

Example: One Of

- Algorithm 3 (binary search):

```
OneOf(A, l, r, x)
    IF l>r
        RETURN False
    value = A[(l+r)/2]
    IF value = x
        RETURN True
    ELSE IF value > x
        RETURN OneOf(A, l, (l+r)/2-1, x)
    ELSE
        RETURN OneOf(A, (l+r)/2+1, r, x)
```

Discussion

- Which algorithm is generally faster?
  - Algorithm 1 or 2?
  - Algorithm 2 or 3?
- Describe an input instance (A, x) such that:
  - Algorithm 1 is the fastest of all
  - Algorithm 2 is the fastest of all
  - Algorithm 3 is the fastest of all

Assumption for Algorithm Analysis

- We only analyze correct algorithms
  - Correct algorithms
    - For every input instance, halt with the correct output
  - Incorrect algorithms
    - Might not halt at all on some input instances
    - Might halt with a wrong answer

Algorithm Analysis - What

- Algorithm analysis predicts the resources that an algorithm requires
  - Memory
  - Computational time (**Efficiency**)
  - Communication bandwidth
  - Power consumption
  - …

Algorithm Analysis - What

- Factors affecting the computational time
  - Computer
  - Compiler
  - Algorithm used
  - Input to the algorithm
    - The *input size* (number of items in the input) affects the running time

Algorithm Analysis - What

- Worst-case running time of an algorithm
  - The longest running time for any input of size *n*
  - An upper bound on the running time for any input
    $\Rightarrow$ guarantee that the algorithm will never take longer
  - Example:
    - Search a linked list for a value, and the value is at the end
- Best-case running time
  - The shortest running time for any input of size *n*
- Average-case running time
  - May be difficult to define what “average” means

Worst-Case Cost

is the focus of our analysis

Algorithm Analysis - How

- Time Cost of an algorithm is
  - The total number of basic operations performed
    - Arithmetical operations
    - Logical operations
    - Assignments
    - Return
  - Usually a function related to the input size

$$T(n) = 3n^2 + 5n$$

Example

```
int sum(int n) {
  int partialSum;

  partialSum = 0;
  for(int i=1; i<=n; i++)
    partialSum += i*i*i;
  return partialSum;
}
```

$$sum(n) = \sum_{i=1}^{n} i^3$$

Example

```
int sum(int n) {
    int partialSum;

 1:  partialSum = 0;  ............................ 1
 2:  for(int i=1; i<=n; i++)  .............. 3n+2
 3:    partialSum += i*i*i;  ............... 4n
 4:  return partialSum;  ...................... 1
}
```

Cost Function: $T(n) = 7n + 4$

Side Note

- With modern compilers, all of the three statements below consumes two basic operations: one addition, one assignment

```
i++;

i += 1;

i = i + 1;
```

At the current stage, we will ignore details and focus on the growth rate of the cost. Under our level of granularity,

$$T_1(n) = 6n + 4 \quad \text{and} \quad T_2(n) = n$$

are of the same

GROWTH RATE

Growth Rate

- Describes how fast the time cost increases as the input size increases
- The idea is to establish a relative order among the cost functions
- Applies only **for large n**
- Typical Order Groups (A.K.A. Complexity Class)

Constant Time:	$T(n) = 1$
Logarithmic Time:	$T(n) = \log n$
Polynomial Time:	$T(n) = n$, $T(n) = n^2$
Exponential Time:	$T(n) = 2^n$, $T(n) = 3^n$

An Analogous Example

- If we place these terms in our grading system …

Order Group	Order	Function
PASS	A	90, 92.5, 93.67
	B	75.4, 81, 82.3
	C	63, 62.2, 66.7
	D	51, 53.1, 55.7
FAIL	F	0, 12, 24.5

Comparing the

GROWTH RATE

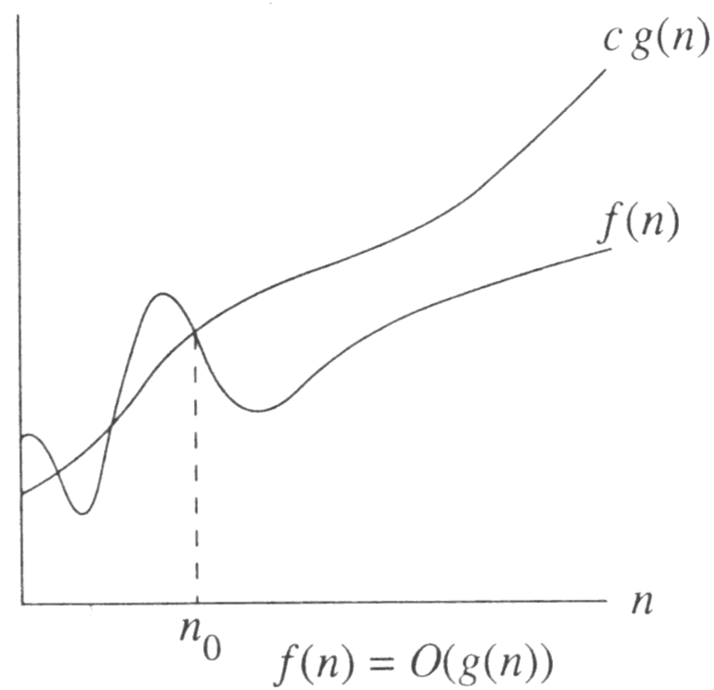
of cost functions

Big-Oh: The Upper Bound

- $f(n) = O(g(n))$
- Definition: There are positive constants c and n_0 such that

  $f(n) \le c\, g(n)$ when $n \ge n_0$

- The growth rate of f(n) is less than or equal to the growth rate of g(n)
  - f(n) grows no faster than g(n) for "large" n
- g(n) is an upper bound of f(n)



Understanding Big-Oh

If the worst-case time cost for an algorithm *A* is

$$g(n) = n$$

Then the time cost for *A* is

$$T(n) = O\big(g(n)\big) = O(n)$$

- Meaning:
  - As input size increases, *A*'s time cost will not grow faster than $g(n)$ does
  - $g(n)$ is the upper bound of *A*'s time cost

Big-Oh: example

- Let $f(N) = 2N^2$. Then
  - $f(N) = O(N^4)$
  - $f(N) = O(N^3)$
  - $f(N) = O(N^2)$ (best answer, asymptotically tight)

- $O(N^2)$: reads “order N-squared” or “Big-Oh N-squared”

Big Oh: more examples

- $N^2 / 2 - 3N = O(N^2)$
- $1 + 4N = O(N)$
- $7N^2 + 10N + 3 = O(N^2) = O(N^3)$
- $\log_{10} N = \log_2 N / \log_2 10 = O(\log_2 N) = O(\log N)$
- $\sin N = O(1)$; $10 = O(1)$, $10^{10} = O(1)$
- $\sum_{i=1}^{N} i \le N \cdot N = O(N^2)$

  $\sum_{i=1}^{N} i^2 \le N \cdot N^2 = O(N^3)$
- $\log N + N = O(N)$
- $\log^k N = O(N)$ for any constant k
- N is $O(2^N)$ but 2^N is not $O(N)$
- 2^N is $O(3^N)$ but 3^N is not $O(2^N)$

Math Review: logarithmic functions

$$x^a = b \quad \mathit{iff} \quad \log_x b = a$$

$$\log ab = \log a + \log b$$

$$\log_a b = \frac{\log_m b}{\log_m a}$$

$$\log a^b = b \log a$$

$$a^{\log n} = n^{\log a}$$

$$\log^b a = (\log a)^b \neq \log a^b$$

$$\frac{d \log_e x}{dx} = \frac{1}{x}$$

Some Rules

When considering the growth rate of a function using Big-Oh

- Ignore the lower order terms and the coefficients of the highest-order term
- No need to specify the base of logarithm
  - Changing the base from one constant to another changes the value of the logarithm by only a constant factor

- If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then
  - $T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$,
  - $T_1(N) * T_2(N) = O(f(N) * g(N))$

Application of the Rules

$$f(n) = 5n^3 + 4n^2 + 3\log n$$

Coefficient

Lower Order Item

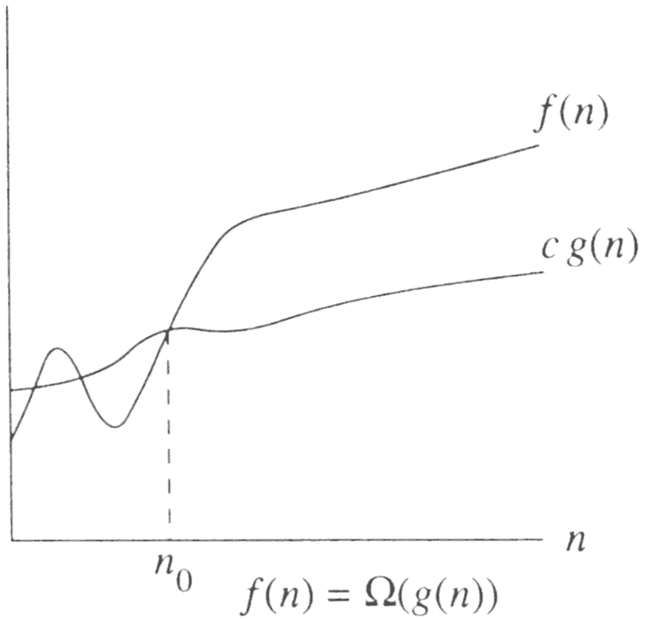
Lower Order Item

Therefore, $f(n) = O(n^3)$

Big-Omega: The Lower Bound

- $f(n) = \Omega(g(n))$
- Definition: There are positive constants c and n_0 such that

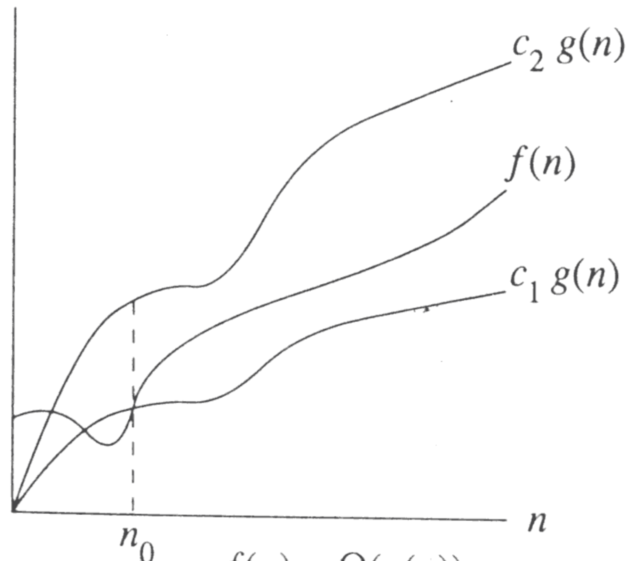
  $f(n) \geq c\, g(n)$ when $n \geq n_0$
- The growth rate of f(n) is greater than or equal to the growth rate of g(n).
- g(n) is a lower bound of f(n)



Big-Omega: examples

- Let $f(N) = 2N^2$. Then
  - $f(N) = \Omega(N)$
  - $f(N) = \Omega(N^2)$ (best answer)

Big-Theta: Tight Bound



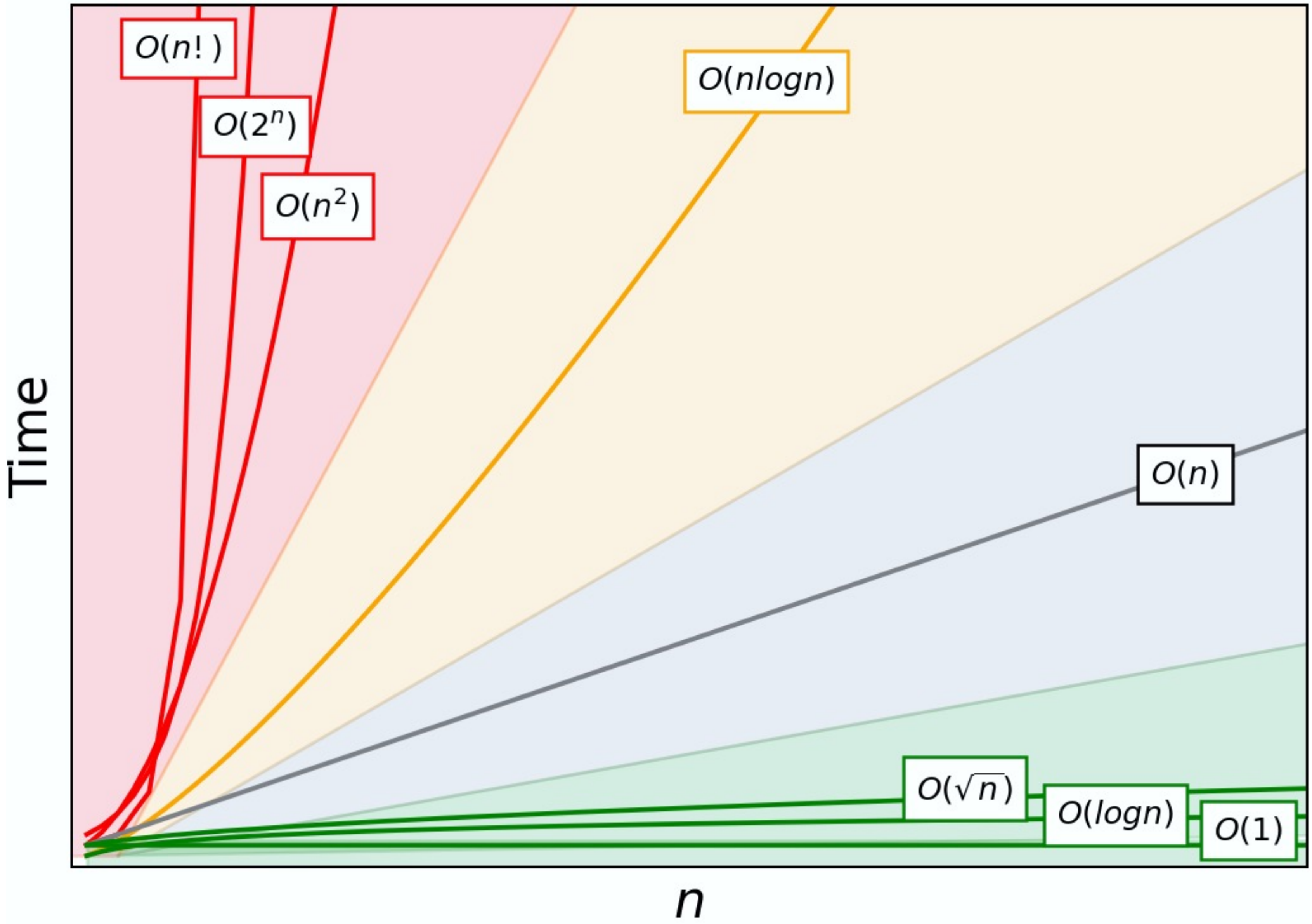
- $f(n) = \Theta(g(n))$ iff.

  $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- The growth rate of f(n) equals that of g(n)
- Big-Theta means the bound is the tightest possible

Some rules

- If T(N) is a polynomial of degree k, then $T(N) = \Theta(N^k)$.
- For logarithmic functions, $T(\log_m N) = \Theta(\log N)$.

Typical Growth Rates



Growth rates …

- Doubling the input size
  - $f(N) = c \Rightarrow f(2N) = f(N) = c$
  - $f(N) = \log N \Rightarrow f(2N) = f(N) + \log 2$
  - $f(N) = N \Rightarrow f(2N) = 2\, f(N)$
  - $f(N) = N^2 \Rightarrow f(2N) = 4\, f(N)$
  - $f(N) = N^3 \Rightarrow f(2N) = 8\, f(N)$
  - $f(N) = 2^N \Rightarrow f(2N) = f^2(N)$
- Advantages of algorithm analysis
  - To eliminate bad algorithms early
  - pinpoints the bottlenecks, which are worth coding carefully

Visualization

- Visualization and Comparison of Sorting Algorithms
- Algorithms used:

Selection Sort	Shell Sort	Insertion Sort
Merge Sort	Quick Sort	Heap Sort
Bubble Sort	Comb Sort	Cocktail Sort

- Introduction of Bubble, Insertion and Quick Sort

Using L' Hopital's rule

- L'Hopital's rule
  - If $\lim_{n\to\infty} f(N) = \infty$ and $\lim_{n\to\infty} g(N) = \infty$

    then $\lim_{n\to\infty} \frac{f(N)}{g(N)} = \lim_{n\to\infty} \frac{f'(N)}{g'(N)}$
- Determine the relative growth rates (using L'Hopital's rule if necessary)
  - compute $\lim_{n\to\infty} \frac{f(N)}{g(N)}$
  - if 0: f(N) = O(g(N)) and f(N) is not Θ(g(N))
  - if constant ≠ 0: f(N) = Θ(g(N))
  - if ∞: f(N) = Ω(g(N)) and f(N) is not Θ(g(N))
  - limit oscillates: no relation

HOW TO DETERMINE GROWTH RATE?

General Rules 1

- `for` loops
  - at most the running time of the statements inside the for-loop (including tests) times the number of iterations.
- Nested `for` loops

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        k++;
```

  - the running time of the statement multiplied by the product of the sizes of all the for-loops.
  - $O(n^2)$

General Rules 2

- Consecutive statements

```
for(i=0; i<n; i++)
    k++;
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        k++;
```

  - These just add up
  - $O(N) + O(N^2) = O(N^2)$

- **`if(E) S1 else S2`**
  - never more than the running time of the test E plus the larger of the running times of S1 and S2.

General Rules 3

- Recursions

```
int sum(int n) {
    if(n<=0)
        return 0;
    return n + sum(n-1);
}
```

  - Find out the recurrence relation between cost functions of different inputs

$$T(n) = \begin{cases} T(n-1) + O(1), & n > 0 \\ O(1), & n \leq 0 \end{cases}$$

  - Then solve the recurrence relation.

$$T(n) = O(n)$$

Appendix: Solving Recurrence Relation

$$T(n) = \begin{cases} T(n-1) + O(1), & n > 0 \\ O(1), & n \leq 0 \end{cases}$$

$$\begin{aligned}
T(n) &= T(n-1) + O(1) \\
&= T(n-1) + 1 \\
&= T(n-2) + 1 + 1 \\
&= T(n-2) + 2 \\
&= T(n-3) + 3 \\
&= \ldots \\
&= T(n-i) + i
\end{aligned}$$

Let i=n:

$$\begin{aligned}
T(n) &= T(n-n) + n \\
&= T(0) + n \\
&= O(n)
\end{aligned}$$