DESIGN AND ANANYSIS ALGORITHMS

LECTURE 2

Analysis of Algorithm

Reference links:

Prof. Xukai Zou, Computer Science Dept.,

Indiana University Purdue University Indianapolis

http://cs.iupui.edu/~xzou/

Prof. Pasi Fränti, School of Computing,

University of Eastern Finland

http://cs.uef.fi/pages/franti/asa/

Anany's book Chapter 2, page 41.

Review the Previous Lesson

- Some key concepts
  - Problem; Algorithm; Data Structure; Program.
- About Complexity and Analysis of Algorithm
- Turing Machine
  - Description; Structure; and Operation
  - Formal definition ➡ Algorithm
- Primitive Recursive Function
  - Basic primitive recursive functions
  - Composition; Primitive recursion

➡ Computable functions

Lecture outline

- The Analysis Framework
- Analyze the complexity of Algorithm
- Prove the correctness of Algorithm
- Exercises

The Analysis Framework

- ✓ Goal of the Analysis of Algorithm
- ✓ Input Data Size and Basic Operation
- ✓ Orders of Growth
- ✓ Types of analysis

Goal of the Analysis of Algorithm

- How to belive the algorithm?
  - Proving the Correctness.
- How to compare the algorithms?
  - Assessing the Effectiveness.

Effectiveness: Two kinds of algorithm efficiency

- Time efficiency - how fast the algorithm runs?
  ⇨ Time complexity – Độ phức tạp thời gian.
- Space efficiency - how much extra memory it uses?
  ⇨ Space complexity – Độ phức tạp không gian.

Goal of the Analysis of Algorithm

- In theoretical analysis of algorithms, the complexity is common estimated in the asymptotic sense, i.e., to estimate the complexity as function for arbitrarily large input.

  Trong phân tích lý thuyết thuật toán, độ phức tạp thường được ước lượng theo nghĩa tiệm cận, tức là ước tính độ phức tạp như một hàm đối với dữ liệu đầu vào có kích thước lớn tùy ý.

Goal of the Analysis of Algorithm

- Theoretical Analysis of Time Complexity:
  - Time complexity is determined by the number of repetitions of the basic operations as a function of the input data size.

n: Input data size

$$T(n) \approx c_{op} C(n)$$

$T(n)$: Running time

c_{op}: Execution time for basic operation

$C(n)$: Number of times basic operation is executed

Input Data Size and Basic Operation

- Input Data Size: number of elements or value of input
- Basic Operation: compare, arithmetic operation, element traversing...

Problem	Input size	Basic operation
Search for key in a list	Number of items in list *n*	Key comparison
Multiply two matrices of floating point number	Dimensions of matrices	Floating point multiplication
Compute a^n	Value of *n*	Floating point multiplication
Find the path on a graph	Size of graph (Number of vertices, number of edges)	Visiting a vertice or traversing an edge

Goal of the Analysis of Algorithm

- Theoretical Analysis of Time Complexity:
  - Example with Insertion Sort

Instruction	Running Time
`InsertionSort(A, n) {`	
`for i = 2 to n {`	$c_1 n$
`key = A[i]`	$c_2(n\text{-}1)$
`j = i - 1;`	$c_3(n\text{-}1)$
`while (j > 0) and (A[j] > key) {`	$c_4 T$
`A[j+1] = A[j]`	$c_5(T\text{-}(n\text{-}1))$
`j = j - 1`	$c_6(T\text{-}(n\text{-}1))$
`}`	
`A[j+1] = key`	$c_7(n\text{-}1)$
`}`	
`}`	

$T = t_2 + t_3 + \ldots + t_n$ where t_i is the times the instructions belong **while** statement called at step i.

Goal of the Analysis of Algorithm

- Theoretical Analysis of Time Complexity:
  - Example with Insertion Sort

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 T + c_5(T-(n-1)) + c_6(T-(n-1)) + c_7(n-1)$$
$$= c_8 T + c_9 n + c_{10}$$

$$\textit{where } T = t_2 + t_3 + \ldots + t_n$$

[Anany's book, section 2.3]

Orders of Growth

$$T(n) \approx c_{op} C(n)$$

- Consider the affect of the factors to the growth of $T(n)$

  Two question:

  - How much faster will algorithm run on computer that is twice as fast?
  - How much longer does it take to solve problem of double input size?

Orders of Growth

- How much faster will algorithm run on computer that is twice as fast?

  $T_1(n) \approx c_{op1}C(n)$

  $T_2(n) \approx c_{op2}C(n) \approx 1/2c_{op1}C(n)$ $\Rightarrow$ $T_2(n) \approx 1/2T_1(n)$

  - c_{op} only affect to $T(n)$ as a multiplication coefficient

Orders of Growth

- How much longer does it take to solve problem of double input size?

  Assume 1: $C(n) = n$

  $$T(n) \approx c_{op} n$$
  $$T(2n) \approx c_{op} 2*n \quad \Rightarrow \quad T(2n) \approx 2T(n)$$

  Assume 2:

  $$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$$

  $$\Rightarrow \quad \frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4.$$

- *C*(*n*) affect to *T*(*n*) by the degree of *n* - **Orders of Growth**

Orders of Growth

- Value of several functions with diffrence orders of growth

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

[Anany's book, page 46]

Orders of Growth

- Basic Efficiency Classes of Growth functions

Time Complexity	Name	Description
1	Constant	Whatever is the input size n, these functions take a constant amount of time.
$logn$	Logarithmic	These are slower growing than even linear functions.
n	Linear	These functions grow linearly with the input size n.
$nlogn$	Linear Logarithmic	Faster growing than linear but slower than quadratic.
n^2	Quadratic	These functions grow faster than the linear logarithmic functions.
n^3	Cubic	Faster growing than quadratic but slower than exponential.
2^n	Exponential	Faster than all of the functions mentioned here except the factorial functions.
$n!$	Factorial	Fastest growing than all these functions mentioned here.

Orders of Growth

- Rates of Growth Fun: Exponetial function

Volume of a grain: $2mm^3$, Total $V=2^{65}$ mm^3 ≈ 12.000 Giza pyramids

http://mathgardenblog.blogspot.com/2015/04/chess.html

Type of Analysis

- Best-case, Average-case, Worst-case

  For some algorithms efficiency depends on type of input:

  - Worst case: W(*n*) – maximum over inputs of size *n*
  - Best case: B(*n*) – minimum over inputs of size *n*
  - Average case: A(*n*) – “average” over inputs of size *n*
    - Number of times the basic operation be execute on typical input
    - NOT the average of worst and best case
    - Expected number of basic operations repetitions considered as a random variable under some assumption about the probability distribution of all possible inputs of size n.

Analyze the complexity of Algorithm

- ✓ Asymptotic Notations
- ✓ Establishing rate of growth relation

Asymptotic Notations

- Asymptotic notations are syntax for presenting the upper and lower bounds of algorithm complexity.
  - Best case but What is the lower bound?
  - Worst case but What is the upper bound?
  - Average case but What is the interval bound?
- A way of comparing functions that ignores constant factors and small input sizes.
- Three main Asymptotic notations:
  - O - Big-oh
  - Ω - Big-omega
  - Θ - Big-theta

Asymptotic Notations

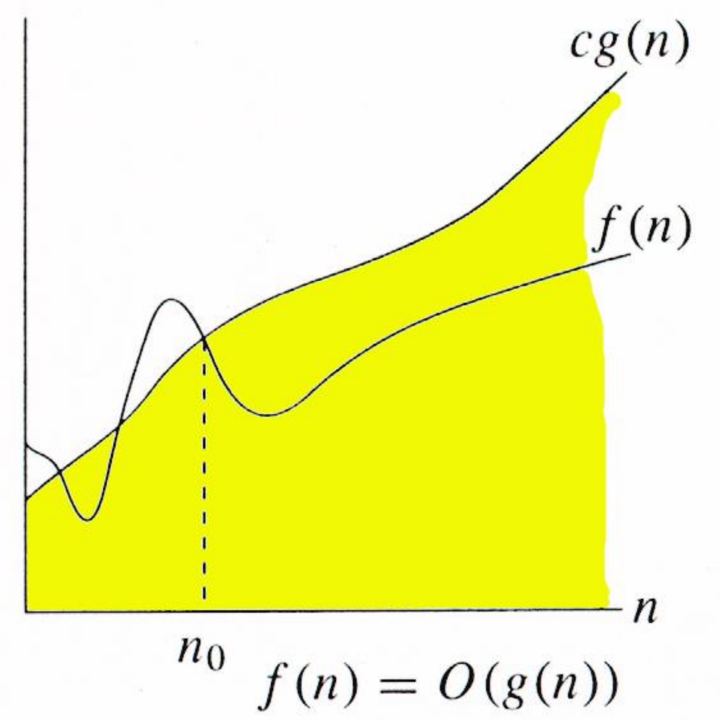
- O - Big-oh (đọc là O lớn – Tiệm cận trên)
  - Definition:

    $f(n) = O(g(n))$ if exist c, n_0 such

    $f(n) \leq c.g(n)$ for all $n \geq n_0$
  - Meaning:

    Class functions $f(n)$ grow

    no faster than $g(n)$



O notation used in worst case evaluation, so interested in the smallest function.

Asymptotic Notations

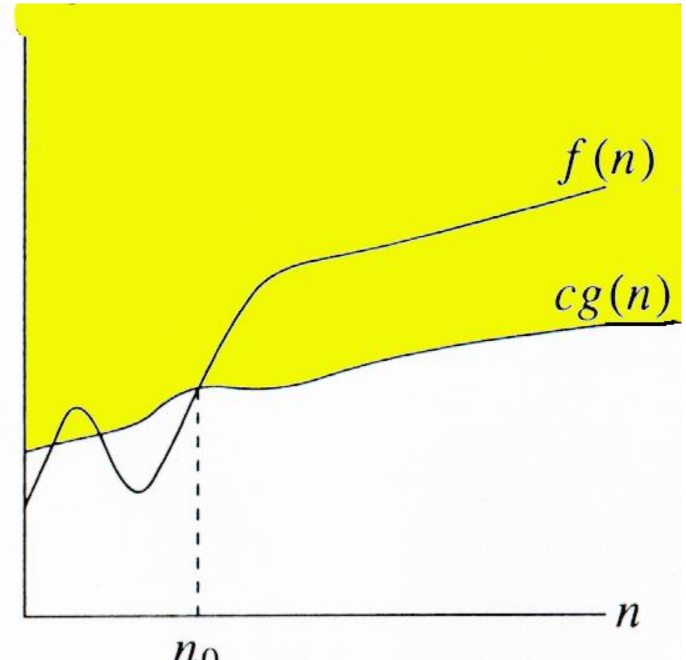
- Ω - Big-omega (đọc là Omega lớn – tiệm cận dưới)
  - Definition:

    $f(n) = \Omega(g(n))$ if exist c, n_0 such

    $f(n) \geq c.g(n)$ for all $n \geq n_0$
  - Meaning:

    Class functions $f(n)$ grow

    at least as fast as $g(n)$



$f(n) = \Omega(g(n))$

Ω notation used in best case evaluation, so interested in the biggest function.

Asymptotic Notations

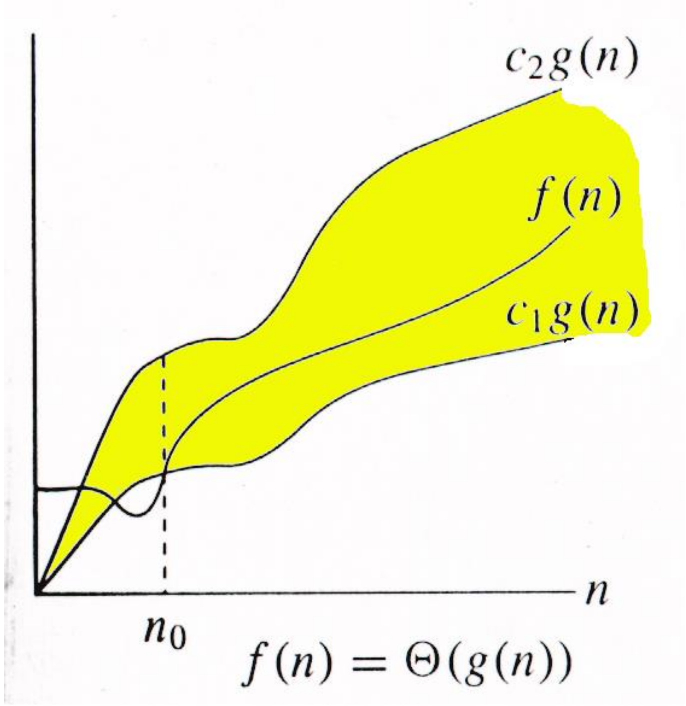
- Θ - Big-theta (đọc là Theta lớn - tiệm cận chặt)
  - Definition:

    $f(n) = \Omega(g(n))$ if exist c_1, c_2, n_0 such

    $c_1.g(n) \leq f(n) \leq c_2.g(n)$ for all $n \geq n_0$
  - Meaning:

    Class functions $f(n)$ grow

    at same rate as $g(n)$



Θ notation used in average case evaluation,
Combined with O, Ω for expressing complexity of algorithm .

Asymptotic Notations

- Example
  - Time complexity of *Insertion Sort* is $O(n^2)$, isn't it?

$$f(n) = an^2 + bn + c = O(n^2)$$

  - Prove:

$$
\begin{aligned}
f(n) &= an^2 + bn + c \\
&\le (a + b + c)n^2 + (a + b + c)n + (a + b + c) \\
&\le 3(a + b + c)n^2 \quad \text{with } n \ge 1
\end{aligned}
$$

Choose $c' = 3(a + b + c)$, $n_0 = 1$ ⇨ $f(n) \le c'.n^2$

or $f(n) = O(n^2)$ ■

- Is $f(n) = an^2+bn+c = O(n^3)$? - Yes, but need the smallest.

- More examples in Anany's book page 53-55

Useful Property of the Asymptotic Notations

- Theorem:

  If $t_1(n) = O(g_1(n))$ and $t_2(n) = O(g_2(n))$, then

  $$t_1(n) + t_2(n) = O(\max\{g_1(n), g_2(n)\}).$$

  - Prove: See [Anany's book page 55].
  - Property useful in analyzing algorithms that comprise two consecutively executed parts.

Useful Property of the Asymptotic Notations

- Other properties of asymptotic relationship:

Transitivity:

$$f(n) = \Theta(g(n)) \;\&\; g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$

$$f(n) = O(g(n)) \;\&\; g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

$$f(n) = \Omega(g(n)) \;\&\; g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$

Reflexivity:

$$f(n) = \Theta(f(n))$$

$$f(n) = O(f(n))$$

$$f(n) = \Omega(f(n))$$

Symmetry and Transpose Symmetry:

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

Establishing rate of growth relation

- Establishing rate of growth relation
  - Given 2 complexity functions f(n) và g(n). Lets determine $f(n) = *(g(n))$ with * is O or Ω or Θ?
- Three methods

Using mathematical definition:

Find constants c, n_0 satisfy the conditions

Using inductive proof:

Example: $\log n = O(n)$ or $\log(n) \leq c.n$

Base: $n = 1 \Rightarrow 0 < 1$ - is true

Inductive step:

Assume $\log(n) \leq n$ when $n>1$

Then $\log(n+1) \leq \log(n+n) = \log(2n) = \log n + 1 \leq n+1$ ■

Using limits (when n $\rightarrow\infty$)

Establishing rate of growth relation

- **Using limits** (when $n \to \infty$)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \Rightarrow f(n) = O(g(n)) \\ \infty & \Rightarrow f(n) = \Omega(g(n)) \\ const & \Rightarrow f(n) = \Theta(g(n)) \\ unknow & \Rightarrow \text{no ralation} \end{cases}$$

Example:

$$Given \quad f(n) = n\sqrt{n} \quad and \quad g(n) = n^2 - n$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n\sqrt{n}}{n^2 - n} = \lim_{n\to\infty} \frac{\sqrt{n}}{n-1} = 0$$

$$\Rightarrow f(n) = O(g(n))$$

More examples [Anany's book page 57]

Establishing rate of growth relation

- Calculus techniques for computing limits
  - L'Hôpital's rule

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = \lim_{n \to \infty} \frac{t'(n)}{g'(n)}$$

  - Stirling's formula

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \text{ for large values of } n.$$

Establishing rate of growth relation

- Basic Efficiency Classes of Growth functions

Time Complexity	Name	Description
1	Constant	Whatever is the input size n, these functions take a constant amount of time.
$logn$	Logarithmic	These are slower growing than even linear functions.
n	Linear	These functions grow linearly with the input size n.
$nlogn$	Linear Logarithmic	Faster growing than linear but slower than quadratic.
n^2	Quadratic	These functions grow faster than the linear logarithmic functions.
n^3	Cubic	Faster growing than quadratic but slower than exponential.
2^n	Exponential	Faster than all of the functions mentioned here except the factorial functions.
$n!$	Factorial	Fastest growing than all these functions mentioned here.

Proving Algorithm’s Correctness

- ✓ Algorithm’s correctness
- ✓ Correctness of Recursive algorithm
- ✓ Correctness of Iterative algorithm

Algorithm Correctness

- How do we know that an algorithm works?

  The answer lead the need of checking correctness.

- Logical methods of checking correctness
  - Testing: Try the algorithm on sample inputs.

    Testing may not find obscure bugs.

    Using tests alone can be dangerous.
  - Correctness proof: Prove mathematically

    Correctness proofs can also contain bugs.

    Use a combination of testing and correctness proof.

Correctness of Recursive algorithm

- Recursive algorithm

  A **recursive algorithm** is an algorithm which calls itself with "smaller (or simpler)" input values.

- Prove recursive algorithm correctness by induction:
  - Prove by induction on the “size” of the problem.
  - ***Base case*** of recursion is base of induction.
  - ***Inductive assumption***: Assume that the recursive call work correctly with input size *n.*
  - ***General case****:* Use the assumption to prove that the current call works correctly with input size *n*+1.

Correctness of Recursive algorithm

- Example
  - Recursive algorithm find maximum of a list:

    Maximum(n) ≡ *//Find the maximum*

    if (n==1) return (A1)

    else return(max(Maximum(n-1),A_n);

    End.
  - Claim: maximum(n) returns max{$A_1, A_2, \ldots, A_n$} for all $n \geq 1$
  - Proof by induction on n:
    - Base case: $n = 1$, maximum(n) return A_1 as claimed
    - Inductive assumption: maximum(n) return max {$A_1, A_2, \ldots, A_n$}
    - General case: Maximum(n+1) return max(Maximum(n),A_{n+1}) = max($A_1, \ldots, A_n, A_{n+1}$)

Correctness of Iterative algorithm

- Iterative algorithm
  - Means non-recursive algorithm
  - Prove iterative algorithm correctness by loop invariant
- Loop invariant
  - Is a logic expression about the variables that remains true every time through the loop. – Biểu thức logic duy trì tính đúng mỗi lần lặp
  - Using loop invariant to prove that the algorithm terminates and computer the correct results. – Sử dụng để chỉ ra thuật toán dừng và cho kết quả.
  - Analyze the algorithm has one loop. In case nested loops, starting at the inner loop. – Nếu vòng lặp lồng nhau sẽ bắt đầu từ vòng lặp trong.

Correctness of Iterative algorithm

- Properties of loop invariant
  - **Initialization** (khởi tạo): The invariant is true before the first iteration.
  - **Maintenance** (duy trì): If the invariant is true at an arbitrary iteration, then it must also be true at the next iteration.
  - **Termination** (kết thúc): The loop always terminates.

  The correctness of the loop (then the algorithm) deduced from the maintaining value true of loop invariant and the termination of the loop.

Correctness of Iterative algorithm

- Example: Non-recursive algorithm find maximum of a list:

  Maximum(n) ≡ *//Find the maximum of the list has n items*

  m = A_1;

  for (i=2;i<=n;i++) if (m<A_i) m = A_i;

  return (m);

  End.

  - Loop invariant: $m_j = \max(A_1,\ldots,A_j)$
  - **Initialization**: $m_1 = A_1 = \max(A_1)$ - is true
  - **Maintenance**: if $m_j = \max(A_1,\ldots,A_j)$, have $m_{j+1} = \max(m_j, A_{j+1}) = \max(A_1,\ldots,A_{j+1})$
  - **Termination**: when i=n+1, after t iteratives (t= n+1-2+1=n)

    $$m_t = \max(A_1,\ldots A_t) = \max(A_1,\ldots A_n)$$

  The loop invariant holds true value and the loop terminates so the correctness of Maximum algorithm is proved.

  See more [Rodney R. Howell, Algorithm: a Top-Down Approach, Chapter 2]

Exercises

- Measuring input's size and orders of growth.
- Establishing rate of growth relation.
- Design and analysis an sorting algorithm.
- More detail in Hw2_AlgorithmAnalysis.doc