Modern Complexity Theory (CS1.405)

Dr. Ashok Kumar Das

IEEE Senior Member
Professor
Center for Security, Theory and Algorithmic Research
International Institute of Information Technology, Hyderabad

E-mail: *ashok.das@iiit.ac.in*
URL: `http://www.iiit.ac.in/people/faculty/ashokkdas`
`https://sites.google.com/view/iitkgpakdas/`

Introduction to Algorithms

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

What is an algorithm?

- An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.
  In addition, all algorithms must satisfy the following criteria:
  1. Input: Zero or more quantities are externally supplied.
  2. Output: At least one quantity is produced.
  3. Definiteness: Each instruction is clear and unambiguous.
  4. Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
  5. Effectiveness: Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation can be definite as in Criteria 3; it also must be feasible.

Reference: E. Horowitz, S. Sahni, and S. Rajasekaran, "Fundamentals of Computer Algorithms," Galgotia Press.

Introduction to Algorithms

What is then a program?

- A program is the expression of an algorithm in a programming language.

Introduction to Algorithms

Important and active research in the study of algorithms

- **How to devise algorithms**
  Creating an algorithm is an art that may never be fully automated. By mastering design strategies, it will become easier for one to devise new and useful algorithms.
- **How to validate algorithms**
  Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs.
  We refer to this process as 'algorithm validation'.

Important and active research in the study of algorithms (Continued...)

- **How to analyze algorithms**
  - This field of study is known as *analysis of algorithms*.
  - As an algorithm is executed, it uses CPU to perform operations and its memory (both immediate and auxiliary) to hold the program and data.
  - Analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage an algorithm requires.
  - This is challenging area which sometimes requires great mathematical skill.

Important and active research in the study of algorithms (Continued...)

- **How to test a program**
  - Testing a program consists of two phases:
    - Debugging: It is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them.
      E. Dijkstra pointed out that “debugging can only point to the presence of errors, but not to their absence”.
    - Profiling or Performance measurement: It is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

Space Complexity

- Space complexity of an algorithm is the amount of memory it needs to run to completion. The space needed by an algorithm is seen to be the sum of two components:
  - **Fixed part:** It is independent of the characteristics (e.g., number, size) of the inputs and outputs.
    This part typically includes the instruction space (i.e., space for the code), space for simple variables and fixed-size component variables (also called aggregate), space for constants, and so on.
  - **Variable part:** It consists of
    - the space needed by component variables whose size is dependent on the particular problem instance being solved
    - the space needed by reference variables (to the extent that this depends on instance characteristics), and
    - the space needed for the recursion stack space (insofar as this space depends on the instance characteristics).

Space Complexity

- The space requirement $S(P)$ of any algorithm (problem) P may therefore be written as

$$S(P) \quad = \quad c + S_P \text{(instance characteristics)}$$

  where c is a constant.
- When analyzing the space complexity of an algorithm, we need to concentrate on estimating S_P (instance characteristics).

Example 1(Space Complexity)

- Consider the following iterative function for sum:

Algorithm: Sum(a, n)

{n is a positive integer and a[] is an array of floating point numbers}

$s = 0.0$;

for $i = 1 \rightarrow n$ **do**

    s = s + a[i];

end for

return s;

Example 1(Space Complexity)

- The problem instances for iterative Sum algorithm are characterized by n, the number of elements to be summed.
- The space needed by a is the space needed by the variables of type array of floats.
  This is at least n words, since a must be large enough to hold the elements to be summed.
- One word is needed for each n, i and s.
- $S_{Sum}(n) \geq n + 3$.

Example 2(Space Complexity)

- Consider the following recursive function for sum:

Algorithm: RSum(a, n)

{n is a positive integer and a[] is an array of floating point numbers}
if $(n \leq 0)$ **then**
  return 0.0;
else
  return RSum(a, n-1) + a[n];
end if

Example 1(Space Complexity)

- The problem instances for recursive Sum algorithm are characterized by n, the number of elements to be summed.
- The recursion stack space includes space for the formal parameters, the local variables, and the return address.
- Assume that return address requires only one word of memory, each call of RSum requires at least 3 words (including space for the values of n, the return address, and a pointer to $a[]$).
- Since the depth of recursion is $n+1$, the recursion stack space needed is $\geq 3(n+1)$
- $S_{RSum}(n) \geq 3(n+1)$.

Time Complexity

- The time $T(P)$ taken by a problem or algorithm P is the sum of the compile time and the run (or execution) time.
- The compile time does not depend on the instance characteristics.
- We assume that a compiled program will run several times without recompilation (like C programs).
- Consequently, we concern with just the run time of a program.
- The run time is denoted by t_P (instance characteristics).
- Express t_P in the form:
  $t_P(n) = c_a.ADD(n) + c_s.SUB(n) + c_m.MUL(n) + c_d.DIV(n) + \ldots$,
  where n denotes the instance characteristics, and c_a, c_s, c_m, c_d, etc., respectively, denote the time needed for an addition, subtraction, multiplication, division, etc.

Time Complexity

- ADD, SUB, MUL, DIV, etc., are functions whose values are the numbers of additions, subtractions, multiplications, divisions, etc., that are performed when the code for P is used on an instance with characteristic n.
- We determine the step count of a program in which the total number of steps contributed by each statement.
- We determine the number of steps per execution (s/e) of the statement and the total number of times (frequency) each statement is executed.
- The s/e of a statement is the amount by which the count changes as a result of the execution of that statement.

Example 3(Time Complexity)

Table: Time complexity for the iterative Sum program: Sum(a, n)

Statement	s/e	frequency	total steps
float Sum(float a[], int n)	0	–	0
$\{$	0	–	0
float $s = 0.0$;	1	1	1
for (int $i = 1; i \leq n; i++$)	1	$n+1$	$n+1$
$\quad s+ = a[i]$;	1	n	n
return s;	1	1	1
$\}$	0	–	0
Total			$2n+3$

Example 4 (Time Complexity)

Table: Time complexity for the recursive Sum program: RSum(a, n)

Statement	s/e	frequency		total steps	
		$n = 0$	$n > 0$	$n = 0$	$n > 0$
float RSum(float a[], int n)	0	–	–	0	0
{	0	–	–	0	0
if ($n \leq 0$)	1	1	1	1	1
return 0.0;	1	1	0	1	0
else return					
$RSum(a, n-1) + a[n]$;	$1 + x$	0	1	0	$1 + x$
}	0	–	–	0	0
Total				2	$2 + x$

Note: $x = t_{RSum}(n-1)$.

Example 5 (Time Complexity)

Table: Time complexity for addition of two matrices $A = (a_{ij})_{m \times n}$ and $B = (b_{ij})_{m \times n}$, and the resultant matrix is $C = A + B = (a_{ij} + b_{ij})_{m \times n}$.

Statement	s/e	frequency	total steps
void AddMatrix(Type a[][SIZE], Type b[][SIZE], Type c[][SIZE], int m, int n)	0	–	0
{	0	–	0
for (int $i = 1; i \leq m; i++$)	1	$m + 1$	$m + 1$
for (int $j = 1; j \leq n; j++$)	1	$m(n + 1)$	$mn + m$
$c[i][j] = a[i][j] + b[i][j];$	1	mn	mn
}	0	–	0
Total			$2mn + 2m + 1$

Asymptotic Notations

Definition (Big "oh" (O))

The function $f(n) = O(g(n))$ (read as "f of n is big oh of g of n") if and only if (iff) there exist positive constants c and n_0 such that $f(n) \leq c.g(n), \forall n \geq n_0$.

- Let $f(n) = 3n + 2$.
  Then $3n + 2 \leq 4n, \forall n \geq 2$.
  Hence, $f(n) = O(n)$.
- Let $g(n) = 1000n^2 + 100n - 6$.
  Then $g(n) = O(n^2)$, as $1000n^2 + 100n - 6 \leq 1001n^2$, for all $n \geq 100$.
- Let $h(n) = 6.2^n + n^2$.
  Then $h(n) = O(2^n)$ as $6.2^n + n^2 \leq 7.2^n$, for all $n \geq 4$.

Asymptotic Notations

- $O(1)$ to mean a computing time that is a constant.
- $O(n)$ is linear
- $O(\log n)$ is logarithmic
- $O(n^2)$ is quadratic
- $O(n^3)$ is cubic
- $O(2^n)$ is exponential
- $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3)$, and $O(2^n)$ are in increasing order.
- $f(n) = O(g(n))$ states that $g(n)$ is an *upper bound* on the value of $f(n)$, for all n, $n \geq n_0$.

Theorem

If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$, *then* $f(n) = O(n^m)$.

Asymptotic Notations

Definition ("Omega" (Ω))

The function $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") if and only if (iff) there exist positive constants c and n_0 such that $f(n) \geq c.g(n), \forall n \geq n_0$.

- Let $f(n) = 3n + 2$.
  Then $3n + 2 \geq 3n, \forall n \geq 1$.
  Hence, $f(n) = \Omega(n)$.
- $f(n) = \Omega(g(n))$ states that $g(n)$ is only a lower bound on $f(n)$.

Theorem

If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ *and* $a_m > 0$, *then* $f(n) = \Omega(n^m)$.

Asymptotic Notations

Definition ("Theta" (Θ))

The function $f(n) = \Theta(g(n))$ (read as "f of n is theta of g of n") if and only if (iff) there exist positive constants c_1, c_2 and n_0 such that $c_1.g(n) \leq f(n) \leq c_2.g(n), \forall n \geq n_0$.
In other words, $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper bound and lower bound on $f(n)$, that is, $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Theorem

If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ *and* $a_m > 0$, *then* $f(n) = \Theta(n^m)$.

Asymptotic Notations

Definition (Little "oh" (o))

The function $f(n) = o(g(n))$ (read as "f of n is little oh of g of n") if and only if (iff) there exist positive constants c and n_0 such that $f(n) < c.g(n), \forall n \geq n_0$.
In other words, the function $f(n) = o(g(n))$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

- Let $f(n) = 3n + 2$.
  Then $f(n) = o(n^2)$,
  since $\lim_{n\to\infty} \frac{3n+2}{n^2} = \lim_{n\to\infty}(\frac{3}{n} + \frac{2}{n^2}) = 0$.
- Let $g(n) = 3n + 2$.
  Then $g(n) = o(n \log n)$, since
  $\lim_{n\to\infty} \frac{3n+2}{n \log n} = \lim_{n\to\infty}(\frac{3}{\log n} + \frac{2}{n \log n}) = 0$.

Asymptotic Notations

Table: Complexity hierarchy and Big-O notations.

Hierarchy	Big-O notations
Constant	$O(1)$
Logarithmic	$O(\log n)$
Polynomial	$O(n^c)$, where c is a constant
Sub-exponential	$O(2^{p(\log n)})$, where $p(\cdot)$ is a polynomial in $\log n$
Exponential	$O(2^n)$
Super-exponential	$O(n^n)$ or $O(2^{2^n})$

Asymptotic Notations

- An algorithm with constant, logarithmic, and polynomial complexity is considered “feasible” for any size of n.
- An algorithm with exponential and super-exponential complexity is considered “infeasible” if n is large.
- An algorithm with sub-exponential complexity is considered “infeasible” if n is very large.

Two broad categories

- **Undecidable Problems**
  - An undecidable problem is a problem for which there is no algorithm that can solve it.
  - Alan Turing proved that the famous “halting problem” is undecidable.
    The halting is stated as follows:
    “Given an input and a Turing machine, there is no algorithm to determine if the machine will eventually halt”.
  - There are several problems in mathematics and computer science that are undecidable.

Two broad categories

- **Decidable Problems**
  - A decidable problem is a problem, if an algorithm can be written to solve it.
  - The corresponding algorithm, however, may or may not be feasible.
  - If a problem can be solved using an algorithm of polynomial complexity or less, it is called a “tractable” problem.
  - If a problem can be solved using an algorithm of exponential complexity, it is called an “intractable” problem.

Complexity theory (in terms of time complexity) divides tractable problems into three classes: P, NP, and coNP.

THE CHURCH-TURING THESIS

The Definition of Algorithm

- Informally speaking, an algorithm is a collection of simple instructions for carrying out some task.
- Alonzo Church used a notational system called the λ-calculus to define algorithms.
- Alan Turing did it with his "machines".
- These two definitions were shown to be equivalent.
- This connection between the informal notion of algorithm and the precise definition to be called the "Church-Turing thesis".

Table: The Church-Turing thesis.

Intuitive notion of algorithms	equals	Turing machine algorithms

End of this lecture