Fundamentals of the Analysis of Algorithm Efficiency

CMSC 142: Design and Analysis of Algorithms

Department of Mathematics and Computer Science
College of Science
University of the Philippines Baguio

Reference: Chapter 2: Section 2.1 to 2.2 of Introduction to Design and Analysis of Algorithms 3rd Edition (Levitin, 2012)

Overview

Analysis Framework

Asymptotic Notation

Basic Efficiency Classes

Analysis of Algorithm Efficiency

Two kinds of efficiency:

- **Time efficiency** indicates how fast an algorithm in question runs.
- **Space efficiency** deals with the extra space the algorithm requires.

Approaches

- Theoretical analysis - based on theoretical laws requiring mathematical framework
- Empirical analysis - ”statistical”, inferred from data

Analysis of Algorithm Efficiency

How to find the time complexity or running time of an algorithm?

Exact Average Analysis

- examining the exact running time
- Machine is needed; use timer functions when running the algorithm for different inputs then observe how much time will it take
- machine-dependent; Need to consider the type of computer, whether it has one or many users at a time, what processor it has, how fast its clock is, how well the compiler optimizes the executable code, etc.
- not the best solution; not at all practical

Analysis of Algorithm Efficiency

How to find the time complexity or running time of an algorithm?

Asymptotic Analysis

- approximate measure of time complexity
- Key idea: run time depends on size of input n and we define function $f(n)$ for the number of instructions executed for input value n
- the growth rate of $f(n)$ with respect to n is of interest
- It might be possible that for smaller input size, the algorithm is better but for larger input size it may not. It is at larger problem sizes that the differences in efficiency between algorithms become particularly prominent.

Analysis Framework

Measuring an Input's Size:

- Almost all algorithms run longer on larger inputs.
- Investigate an algorithm's efficiency as a function of some parameter n, indicating the algorithms input size
- Example of computational problem

(i) searching, sorting
(ii) function evaluation
(iii) matrix multiplication
(iv) graph problem
(v) checking primality of an integer

- The size of inputs for algorithms involving properties of real numbers is the number b of bits on the number $n's$ binary representation: $b = \lfloor \log_2 n \rfloor + 1$.

Analysis Framework

Units for Measuring Running Time:

- Standard unit of time measurement has drawbacks.
- The efficiency is analyzed by determining the number of repetitions of the **basic operation** (*operation that contributes the most towards the running time of the algorithm*) as function of input size.

Problem	Input size measure	Basic Operation
Searching for key in a list of n items	Number of list's items i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer	$\lfloor \log_2 n \rfloor + 1$	Division

Analysis Framework

Consider an algorithm running on an input size n. Define:

- c_{op} - execution time of the algorithm's basic operation on a particular computer
- $C(n)$ - number of times the basic operation needs to be executed for the algorithm

Then the estimated running time $T(n)$ is given by the formula

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

Example

Assuming that $C(n) = \frac{1}{2}n(n-1)$, how much longer will the algorithm run if we double its input size?

Analysis Framework

Remarks

1. The count $C(n)$ does not contain any information about operations that are not basic, and, the count is often computed only approximately.
2. c_{op} is an approximation whose reliability is not always easy to assess.
3. When n is extremely large or very small, the formula $T(n)$ give a reasonable estimate of the algorithm's running time.
4. The formula makes it possible to answer questions such as "How much faster would this algorithm run on a machine that is 10 times faster than the one we have?" and "How much longer will the algorithm run if we double its input size?"

Orders of Growth

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}		

Figure: Values of several functions important for analysis of algorithms

Remark. Algorithms that require an exponential number of operations are practical for solving only problems of small sizes.

Analysis Framework

Worst-Case, Best Case and Average-Case Efficiencies

Consider the algorithm below for sequential search.

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
//              or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Worst-Case

The **worst-case efficiency** of an algorithm is its efficiency for the worst-case input of size n, input for which the algorithm runs the longest among all possible inputs of that size.

The worst-case analysis provides very important information about an algorithms efficiency by bounding its running time from above.

Example

For sequential search, the worst-case is when there are no matching elements or the first matching element happens to be the last one on the list, that is $C_{worst}(n) = n$, where n is the size of the input.

Best-Case

The **best-case efficiency** of an algorithm is its efficiency for the best case input of size n, which is an input of size n for which the algorithm runs the fastest among all possible inputs of that size.

The best case does not mean the smallest input; it means the input of size n for which the algorithm runs the fastest.

Example

For sequential search, the best-case inputs are lists of size n, with their first elements equal to a search key; accordingly $C_{best}(n) = 1$.

Average-Case

The **average-case efficiency** seeks to provide the necessary information about an algorithm's behavior on a "typical" or "random" input.

- determine the number of different groups, say m, into which all possible input sets can be divided
- determine the probability p_i $(0 \leq p_i \leq 1)$ that input will come from each group i
- determine efficiency (running time t_i) for each group

The average case efficiency is: $C_{avg}(n) = \sum_{i=1}^{m} p_i \times t_i$

Example. Sequential Search

Suppose the probability of a sequential search is equal to p and that the probability of the first match occurring in the i^{th} position of the list is the same for every i. Then the average number of key comparisons $C_{avg}(n)$ is:

(i) In a successful search, the probability of the first match occurring in the i^{th} position of the list is p/n for every i and the number of comparisons made by the algorithm is i.

(ii) In the case of an unsuccessful search, the number of comparisons is n with the probability of such a search being $(1-p)$.

So, $C_{avg}(n) = \dfrac{p(n+1)}{2} + n(1-p)$.

Analysis Framework: Summary

- Both time and space efficiencies are measured as functions of the algorithm's input size.
- Time efficiency is measured by counting the number of times the algorithm's basic operation is executed. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.
- The efficiencies of some algorithms may differ significantly for inputs of the same size. For such algorithms, we need to distinguish between the worst-case, average-case, and best-case efficiencies.
- The framework's primary interest lies in the **order of growth** of the algorithm's running time (extra memory units consumed) as its input size goes to infinity.

Short Exercise

Write an algorithm that finds the median value of three distinct integers x, y, z using only operation key comparison (no sorting).

(i) The input for this algorithm falls into how many groups?

(ii) What is the best case and worst case for the algorithm?

(iii) What is the average case?

Overview

Analysis Framework

Asymptotic Notation

Basic Efficiency Classes

Asymptotic Notation: Big Oh

A function $t(n)$ is said to be in $\boldsymbol{O}(g(n))$, denoted $t(n) \in O(g(n))$, if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \quad \text{for all } n \geq n_0.$$

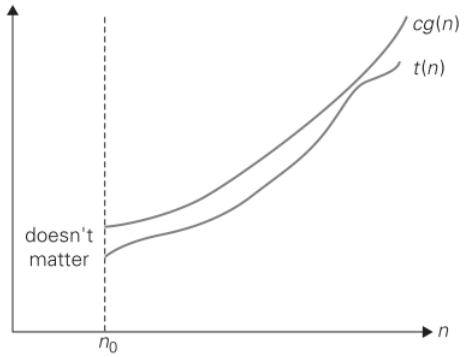

Figure: Big-Oh notation: $t(n) \in O(g(n))$.

Asymptotic Notation: Big Omega

A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \quad \text{for all } n \geq n_0.$$

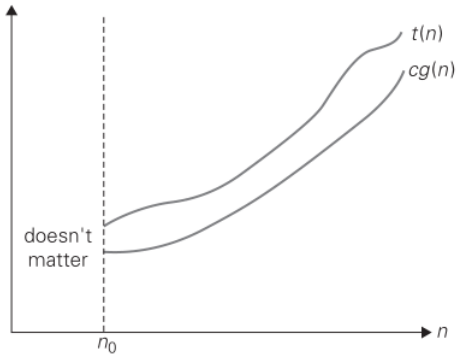

Figure: Big-Omega notation: $t(n) \in \Omega(g(n))$.

Asymptotic Notation: Big Theta

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n)$$

for all $n \geq n_0$.

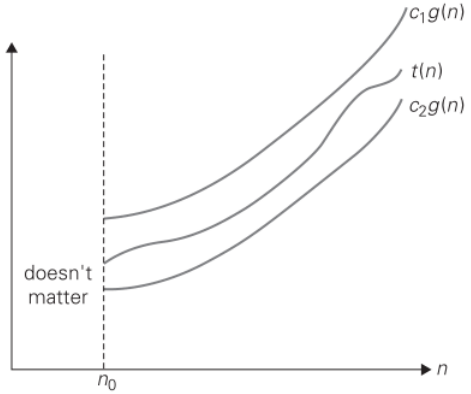

Figure: Big-Theta notation: $t(n) \in \Theta(g(n))$.

Asymptotic Notation

Example. Determine whether the following assertions are true or false.

a. $\dfrac{n(n+1)}{2} \in O(n^3)$

b. $\dfrac{n(n+1)}{2} \in O(n^2)$

c. $\dfrac{n(n+1)}{2} \in \Theta(n^3)$

d. $\dfrac{n(n+1)}{2} \in \Omega(n)$

Example. List the following functions from highest to lowest order

$$2^n, \quad \log n, \quad n + n^2 + 5n^3, \quad n^2, \quad n \log n, \quad n!, \quad \log(\log n), \quad n^3 + \log n$$

Asymptotic Notation

Theorem

If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then

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

(The analogous assertions are true for the Ω and Θ notations as well.)

Example

1. Let $t(n) = 2^{n+1} + 3^{n-1}$. Find $g(n)$ such that $t(n) \in \Theta(g(n))$.
2. **True or False**. If $t(n) \in O(g(n))$ then $g(n) \in \Omega(t(n))$.

Asymptotic Notation

Theorem . Let $t(n)$ and $g(n)$ be any nonnegative functions defined on the set of natural numbers. Then

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{then } t(n) \text{ has a smaller order of growth than } g(n) \\ c > 0 & \text{then } t(n) \text{ has the same order of growth as } g(n) \\ \infty & \text{then } t(n) \text{ has a larger order of growth than } g(n) \end{cases}$$

Stirling's formula: For large values of n, $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

Example. Compare the orders of growth of the following: (a) $\frac{1}{2}n(n-1)$ and n^2, (b) $\log_2 n$ and $\sqrt{n}$, and (c) $n!$ and 2^n.

Overview

Analysis Framework

Asymptotic Notation

Basic Efficiency Classes

Basic Efficiency Classes

The time efficiencies of a large number of algorithms fall into the following classes (*listed in increasing orders of their growth*).

Class	Name	Remarks
1	constant	Short of best-case efficiencies
$\log n$	logarithmic	Algo that cuts problem's size on each iteration
n	linear	Algo that scan a list of size n
$n \log n$	"$n \log n$"	Algo of divide-and-conquer design
n^2	quadratic	Algo with two embedded loops
n^3	cubic	Algo with three embedded loops
2^n	exponential	Algo generating all subsets of an $n-$set
$n!$	factorial	Algo generating permutations of an $n-$set

End of Lecture