CSC 344 – Algorithms and Complexity

Lecture #2 – Analyzing Algorithms and Big O Notation

Analysis of Algorithms

- Issues:
  - Correctness
  - Time Efficiency
  - Space Efficiency
  - Optimality
- Approaches:
  - Theoretical Analysis
  - Empirical Analysis

Analysis of Algorithms - Issues

- Issues:
  - Correctness – *Does it work as advertised?*
  - Time Efficiency – *Are time requirements minimized?*
  - Space Efficiency – *Are space requirements minimized?*
  - Optimality – *Do we have the best balance between minimizing time and space?*

Theoretical Analysis Of Time Efficiency

- Time efficiency is analyzed by determining the number of repetitions of the <u>*basic operation*</u> as a function of <u>*input size*</u>
- <u>*Basic operation*</u>: the operation that contributes most towards the running time of the algorithm

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

Running Time ($T(n)$)

Execution Time For Basic Operation (c_{op})

Number Of Times Basic Operation Is Executed ($C(n)$)

Input Size And Basic Operation Examples

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 *n*	*n*'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Empirical Analysis Of Time Efficiency

- Select a specific (typical) sample of inputs
- Use physical unit of time (e.g., milliseconds)

  or
- Count actual number of basic operation's executions
- Analyze the empirical data

Best-Case, Average-Case, Worst-Case

- For some algorithms efficiency depends on form of input:
  - Worst case: $C_{worst}(n)$ – maximum over inputs of size n
  - Best case: $C_{best}(n)$ – minimum over inputs of size n
  - Average case: $C_{avg}(n)$ – "average" over inputs of size n

Average-Case

- Average case: $C_{avg}(n)$ – "average" over inputs of size n
  - Number of times the basic operation will be executed on typical input
  - NOT the average of worst and best case
  - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs

Example: 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 of 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
```

- Best case?
- Worst case?
- Average case?

Types Of Formulas For Basic Operation's Count

- Exact formula

  e.g., $C(n) = n(n-1)/2$

- Formula indicating order of growth with specific multiplicative constant

  e.g., $C(n) \approx 0.5\, n^2$

- Formula indicating order of growth with unknown multiplicative constant

  e.g., $C(n) \approx cn^2$

Order of Growth

- ***Most important***: Order of growth within a constant multiple as n→∞
- Example:
  - 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?

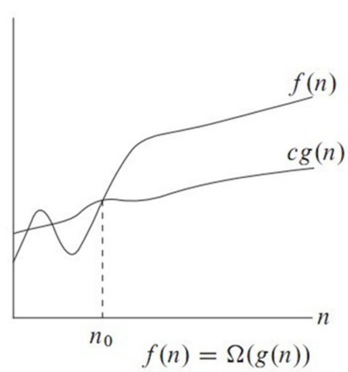
Values of Some Important Functions as n →∞

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

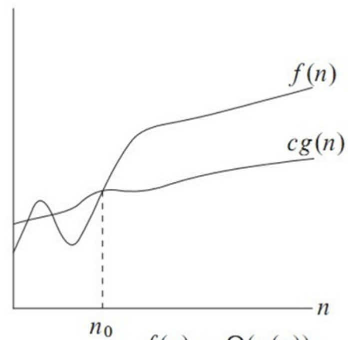
Asymptotic Order Of Growth

- A way of comparing functions that ignores constant factors and small input sizes
  - **O(g(n))** - class of functions f(n) that grow no faster than g(n)
  - **Θ(g(n))** - class of functions f(n) that grow at same rate as g(n)
  - **Ω(g(n))** - class of functions f(n) that grow at least as fast as g(n)

Big *O*

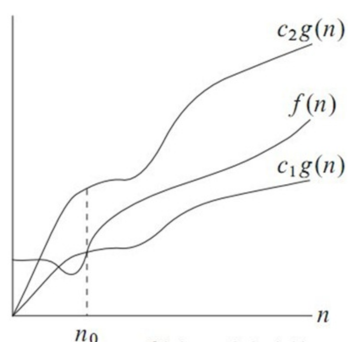


Big Omega



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

Big Theta



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

Establishing Order Of Growth Using The Definition

- Definition: $f(n)$ is in $O(g(n))$ if order of growth of $f(n) \leq$ order of growth of $g(n)$ (within constant multiple), i.e., there exist positive constant c and non-negative integer n_0 such that

$$f(n) \leq c\, g(n) \text{ for every } n \geq n_0$$

- Examples:
  - $10n$ is $O(n^2)$
  - $5n+20$ is $O(n)$

Some Properties Of Asymptotic Order Of Growth

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$
- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

  Note similarity with $a \leq b$
- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$

Establishing Order Of Growth Using Limits

$$\lim_{n\to\infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } \boldsymbol{T(n)} < \text{ order of growth of } \boldsymbol{g(n)} \\ c > 0 & \text{order of growth of } \boldsymbol{T(n)} = \text{ order of growth of } \boldsymbol{g(n)} \\ \infty & \text{order of growth of } \boldsymbol{T(n)} > \text{ order of growth of } \boldsymbol{g(n)} \end{cases}$$

Examples:

- **$10n$ vs. n^2**
- **$n(n+1)/2$ vs. n^2**

L'Hôpital's Rule And Stirling's Formula

- L'Hôpital's rule:

  If $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$ and the derivatives f', g' exist,

  then $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$

  - Example: $\log n$ vs. n
- Stirling's formula: $n! \approx (2\pi n)^{1/2} (n/e)^n$
  - Example: 2^n vs. $n!$

Orders Of Growth Of Some Important Functions

- All logarithmic functions $\log_a n$ belong to the same class $\Theta(\log n)$ no matter what the logarithm's base $a > 1$ is

- All polynomials of the same degree k belong to the same class: $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \in \Theta(n^k)$

- Exponential functions a^n have different orders of growth for different a's

- order $\log n$ < order n^{α} $(\alpha>0)$ < order a^n < order $n!$ < order n^n

Basic Asymptotic Efficiency Classes

1	**constant**
$\log n$	logarithmic
n	linear
$n \log n$	n-log-n or linearithmic
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

Time Efficiency Of Nonrecursive Algorithms

General Plan for Analysis

- Decide on parameter n indicating ***input size***
- Identify algorithm's ***basic operation***
- Determine ***worst***, ***average***, and ***best*** cases for nput of size n
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules

Useful Summation Formulas And Rules

$\Sigma_{l\le i\le u} 1 = 1+1+\cdots+1 = u - l + 1$

In particular, $\Sigma_{1\le i\le u} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1\le i\le n}\, i = 1+2+\cdots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1\le i\le n}\, i^2 = 1^2+2^2+\cdots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0\le i\le n}\, a^i = 1 + a + \cdots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \ne 1$

In particular, $\Sigma_{0\le i\le n}\, 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i$ $\quad \Sigma c a_i = c\Sigma a_i$ $\quad \Sigma_{l\le i\le u} a_i = \Sigma_{l\le i\le m} a_i + \Sigma_{m+1\le i\le u} a_i$

Example 1 - Maximum Element

ALGORITHM *MaxElement*$(A[0..n-1])$
//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval$

Example 2 - Element Uniqueness problem

ALGORITHM *UniqueElements*$(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

Example 3 - Matrix Multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

Example 4: Counting Binary Digits

ALGORITHM *Binary*(n)

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```