Analysis of Algorithms

Chapter 2.1, 2.2, 2.3

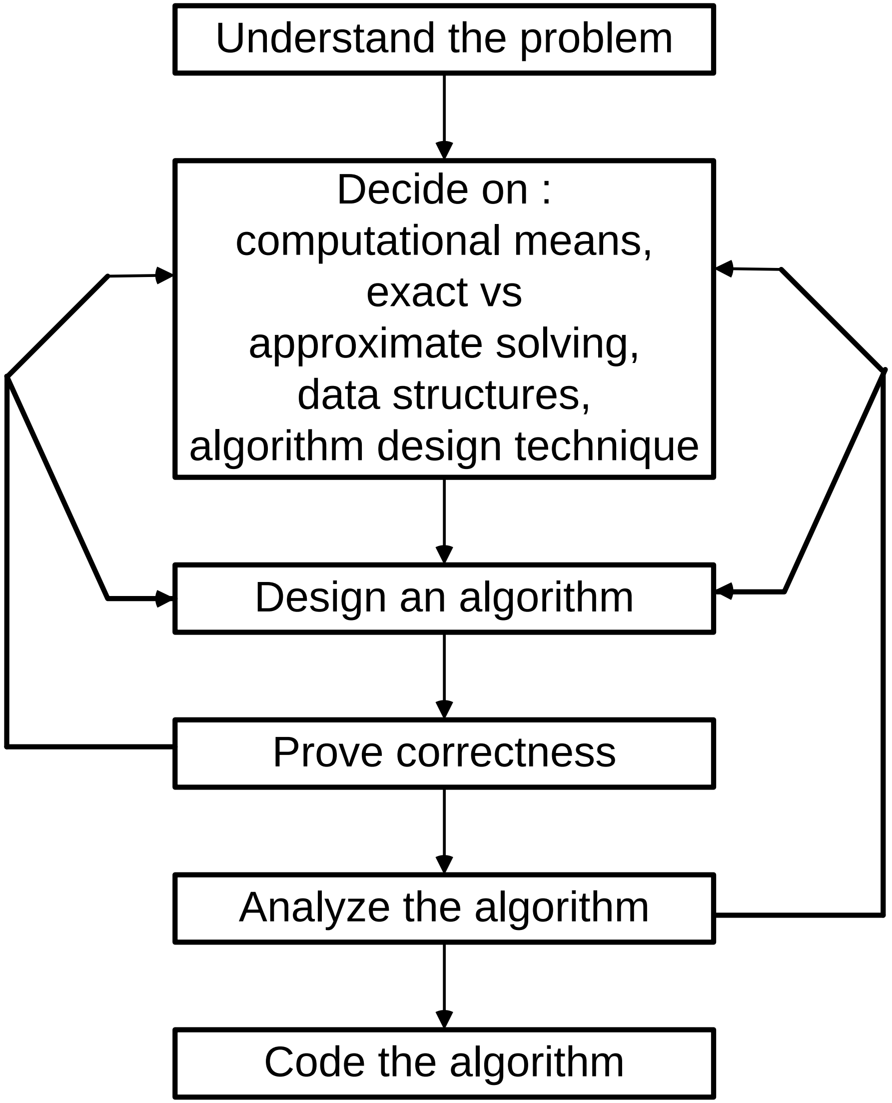
Review of Chapter 1

- Two main issues related to algorithms
  - How to design algorithms
  - How to analyze algorithm efficiency
- Properties of an Algorithm
  - **Effectiveness**
    - Instructions are simple
      - can be carried out by pen and paper
  - **Definiteness**
    - Instructions are clear
      - meaning is unique
  - **Correctness**
    - Algorithm gives the right answer
      - for all possible cases
  - **Finiteness**
    - Algorithm stops in reasonable time
      - produces an output

Donald E. Knuth
Professor Emeritus of The Art of Computer Programming at Stanford University
Knuth has been called the "father" of the analysis of algorithms

A person well-trained in computer science knows how to deal with algorithms: how to construct them, manipulate them, understand them, analyze them. This knowledge is preparation for much more than writing good computer programs; it is a general-purpose mental tool that will be a definite aid to the understanding of other subjects, whether they be chemistry, linguistics, or music, etc. The reason for this may be understood in the following way: It has often been said that a person does not really understand something until after teaching it to someone else. Actually, a person does not *really* understand something until after teaching it to a *computer*, i.e., expressing it as an algorithm . . . An attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way. [Knu96, p. 9]

Algorithm Design Process



Euclid's Algorithm

- **Problem:** Find gcd(m,n), the greatest common divisor of two nonnegative, not both zero integers m and n

- Examples: gcd(60,24) = 12, gcd(60,0) = 60, gcd(0,0) = ?

- Euclid's algorithm is based on repeated application of equality

  $$\gcd(m,n) = \gcd(n, m \bmod n)$$

- until the second number becomes 0, which makes the problem trivial.

  Example: gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

Structured Description of Euclid's Algorithm

- **Step 1** If $n = 0$, return m and stop; otherwise go to Step 2
- **Step 2** Divide m by n and assign the value to the remainder to r
- **Step 3** Assign the value of n to m and the value of r to n. Go to Step 1.

Euclid's Algorithm (Pseudocode)

ALGORITHM *Euclid*(m, n)

//Computes $\gcd(m, n)$ by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n

while $n \neq 0$ **do**
  $r \leftarrow m \bmod n$
  $m \leftarrow n$
  $n \leftarrow r$
return m

Consecutive integer checking algorithm

Step 1 Assign the value of min{m,n} to t

Step 2 Divide m by t. If the remainder is 0, go to Step 3; otherwise, go to Step 4

Step 3 Divide n by t. If the remainder is 0, return t and stop; otherwise, go to Step 4

Step 4 Decrease t by 1 and go to Step 2

Middle-school procedure for computing gcd(m, n)

Step 1 Find the prime factors of m.

Step 2 Find the prime factors of n.

Step 3 Find all the common prime factors

Step 4 Compute the product of all the common prime factors and return it as gcd*(m,n*)

60 = 2 x 2 x 3 x 5

24 = 2 x 2 x 2 x 3

gcd(60, 24) = 2 x 2 x 3 = 12

- *Is this an algorithm?*

Sieve of Eratosthenes

- **A simple Algorithm Generating Consecutive Primes Not Exceeding Any Given Integer n: Sieve of Eratosthenes**
- Example:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3	x	5	x	7	x	9	x	11	x	13	x	15	x	17	x	19	x	21	x	23	x	**25**
2	**3**		5		7		x		11		13		x		17		19		x		23		**25**
2	**3**		**5**		**7**				**11**		**13**				**17**		**19**				**23**		**x**

Sieve of Eratosthenes

ALGORITHM $Sieve(n)$

```
//Implements the sieve of Eratosthenes
//Input: An integer n ≥ 2
//Output: Array L of all prime numbers less than or equal to n
for p ← 2 to n do A[p] ← p
for p ← 2 to ⌊√n⌋ do    //see note before pseudocode
    if A[p] ≠ 0   //p hasn't been eliminated on previous passes
        j ← p * p
        while j ≤ n do
            A[j] ← 0    //mark element as eliminated
            j ← j + p
//copy the remaining elements of A to array L of the primes
i ← 0
for p ← 2 to n do
    if A[p] ≠ 0
        L[i] ← A[p]
        i ← i + 1
return L
```

ROAD MAP

- **Analysis of algorithms**
- Running time functions
- Mathematical Analysis of Nonrecursive Algorithms
- Recurrence Relations
  - Exact Solution
    - Forward substitution
    - Backward substitution
    - Methods similar to those used in solving differential equations
  - Asymptotic Solution
    - Master theorem

Analysis of algorithms

- Issues:
  - correctness
  - time efficiency
  - space efficiency
  - optimality

- Approaches:
  - theoretical analysis
  - empirical analysis

Analysis of Algorithms

- Study complexity of an algorithm
  - How good is the algorithm?
  - How is it when compared with other algorithms?
  - Is it the best that can be done?

Analysis of Algorithms

- Complexities
  - Space
    - Number of bits
    - Number of elements
  - Time
    - Number of operations
      - Depends on model
      - RAM

Run-Time Analysis of Algorithms

- Algorithm complexity is investigated as a function of some parameter *n* indicating problem's size

- Time complexity, T(*n*), is can be computed as the number of times the algorithm's most important operation -- called its <u>basic operation</u> -- is executed

- Space complexity, S(*n*), is usually computed as the size of memory space used during an execution of the algorithm

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*

- *Basic operation*: the operation that contributes most towards the running time of the algorithm

input size

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

running time

execution time
for basic operation

Number of times
basic operation is
executed

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

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 $n \to \infty$

- 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	$3.3 \cdot 10^1$	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}		

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

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
  - 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

Types of Complexities

- Worst case

$$T(n) = max_{|I|=n}\{T(I)\}$$

- Average case

$$T(n) = \sum_{|I|=n} T(I).\Pr ob(I)$$

- Best case

$$T(n) = min_{|I|=n}\{T(I)\}$$

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
$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
    $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

- Worst case
- Best case
- Average case

Sequential search

Algorithm Complexity:

- Best case

  A[1] = key
- Worst case

  A[i] ≠ key   for any key
  - time is proportional to the number of elements
  - time complexity of linear search is O(n)
- Average case ?
  - if any key is equally likely ~ n/2

ROAD MAP

- Analysis of algorithms
- **Running time functions**
- Mathematical Analysis of Nonrecursive Algorithms
- Recurrence Relations
  - Exact Solution
    - Forward substitution
    - Backward substitution
    - Methods similar to those used in solving differential equations
  - Asymptotic Solution
    - Master theorem

Running Time Functions

- Definition

  A nondecreasing function is called ***running time function*** if

  $f:Z^+ \rightarrow R$ such that $f(n)>0$ for all $n \geq m$ where m is some positive integer

  $\mathbf{Z^+ = \{ 1, 2, 3, \ldots \}}$

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 <u>no faster</u> than $g(n)$

- $\Theta(g(n))$: class of functions $f(n)$ that grow <u>at same rate</u> as $g(n)$

- $\Omega(g(n))$: class of functions $f(n)$ that grow <u>at least as fast</u> as $g(n)$

Asymptotic notations

O notation

- <u>Definition</u>

  Let f and g are running time functions. We denote ***f(n) = O(g(n))*** if there exists a real constant c and integer m such that

$$f(n) \le c\,(g(n)) \quad \text{for all } n \ge m$$

Big-oh

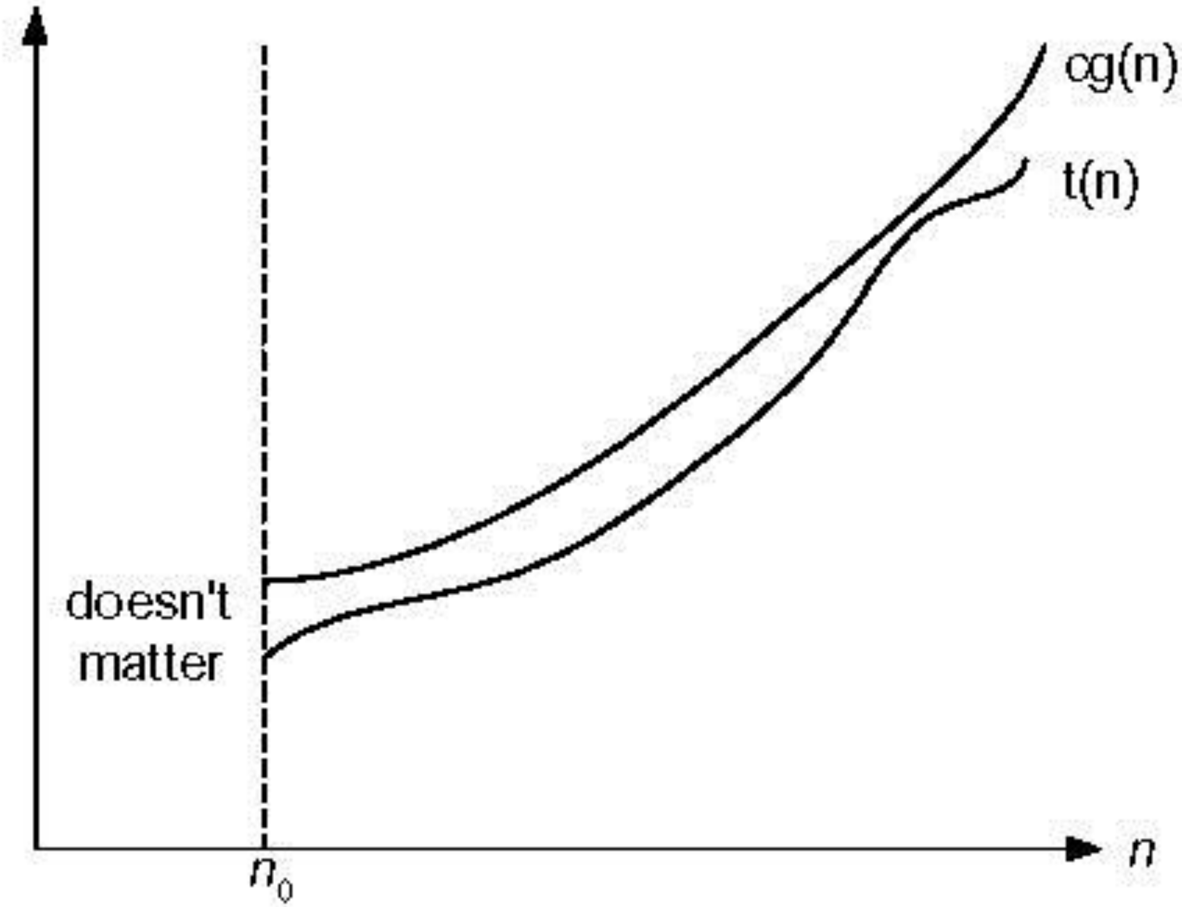


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

O notation

- Ex:
  $7n + 5 = O(n)$

- Ex:
  $10n^2 + 4n + 2 = O(n^2)$

- Ex:
  $7n + 5 = O(n^2)$

- Ex
  $7n + 5 \neq O(1)$

Asymptotic notations

Ω notation

- <u>Definition</u>

  Let f and g are running time functions. We denote ***f(n) = Ω(g(n))*** if there exists a real constant c and integer m such that

$$f(n) \geq c\,(g(n)) \quad \text{for all } n \geq m$$

Big-omega

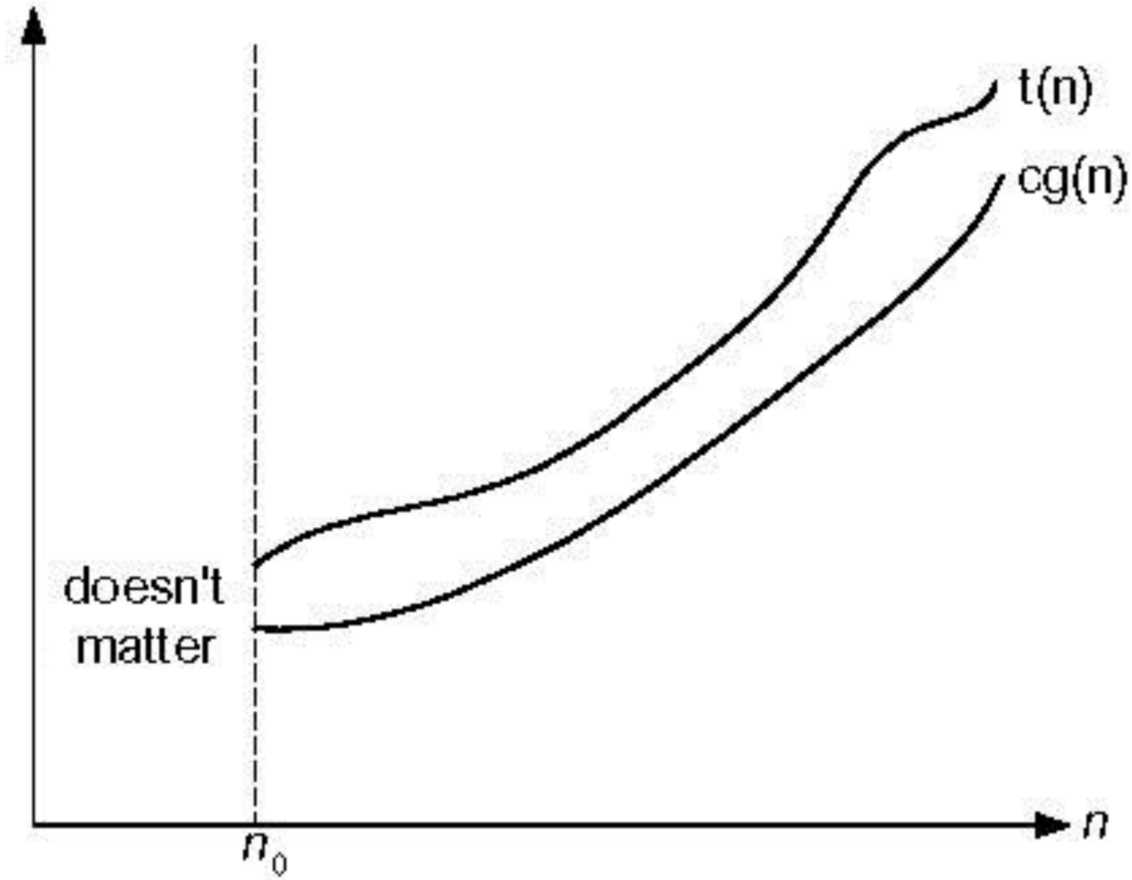


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Ω notation

- Ex:
  $3n + 2 = \Omega(n)$

- Ex:
  $6.2^n + n^2 = \Omega(2^n)$

- Ex:
  $3n - 7 = \Omega(1)$

Asymptotic notations

θ notation

- Definition

  Let f and g are running time functions. We denote ***f(n) = θ(g(n))*** if there exists real constants c_1 and c_2 and integer m such that

$$c_1\, g(n) \leq f(n) \leq c_2\, g(n) \quad \text{for all } n \geq m$$

Big-theta

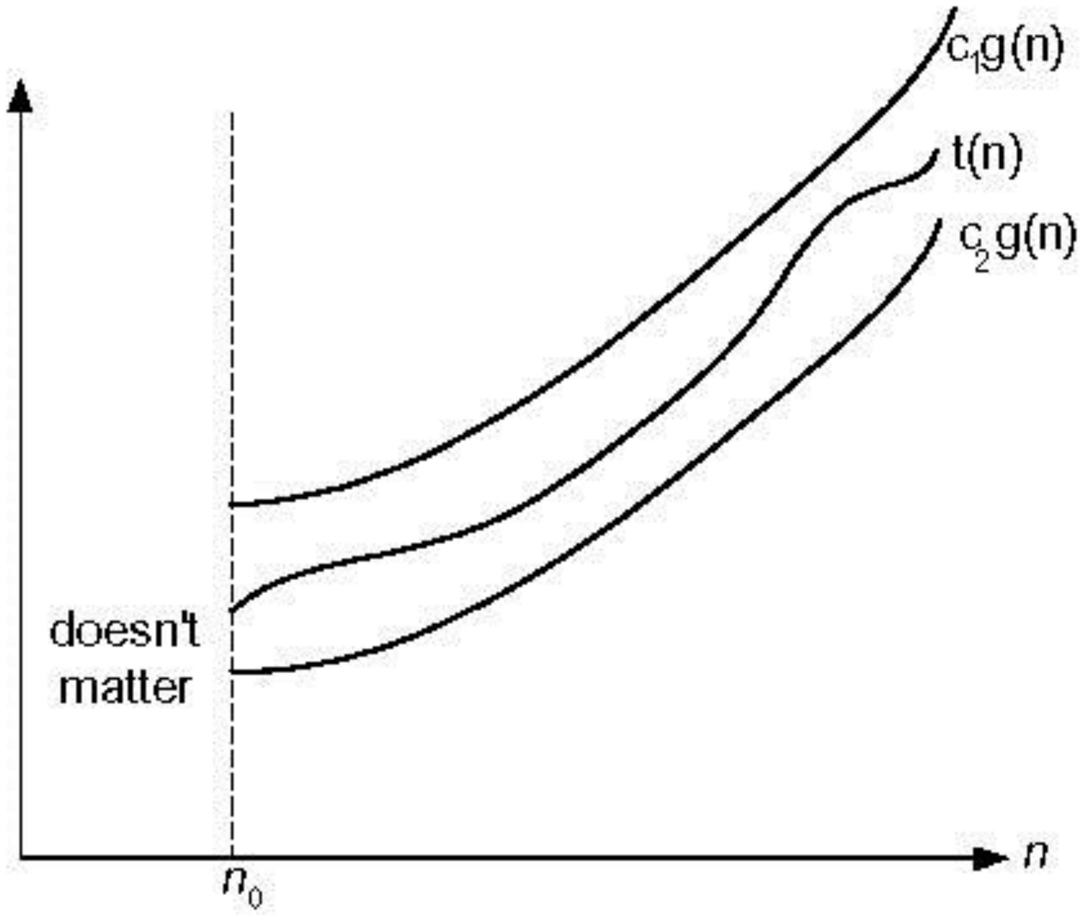


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

θ notation

- Ex:
  $3n + 2 = \theta(n)$

- Ex:
  $10 \log n + 4 = \theta(\log n)$

- Ex:
  $3n + 2 \neq \theta(1)$

Asymptotic notations

- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$
- $f(n) = O(g(n))$
  - g is an upper bound of f
  - f grows no faster than g
- How tight is this bound ?
  - $n=O(n^2)$
  - $n=O(2^n)$
- $f(n) = O(g(n)) \rightarrow g(n) = O(f(n))$?

Some Rules

- Transitivity

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

- Addition

$$f(n) + (g(n)) = O(\max\{f(n), g(n)\})$$

- Polynomials

$$a_0 + a_1 n + \ldots + a_d n^d = O(n^d)$$

Some Rules

- θ is equivalence notation

$$f(n) = \theta(f(n))$$

$$f(n) = \theta(g(n)) \quad \Rightarrow \quad g(n) = \theta(f(n))$$

$$f(n) = \theta(g(n)) \,\&\, g(n) = \theta(h(n)) \Rightarrow f(n) = \theta(h(n))$$

Some Rules

$$f_1(n) = \theta(g(n)) \,\&\, f_2(n) = \theta(g(n))$$
$$\Rightarrow f_1(n) + f_2(n) = \theta(g(n))$$

$$f_1(n) = \theta(g_1(n)) \,\&\, f_2(n) = \theta(g_2(n))$$
$$\Rightarrow f_1(n) + f_2(n) = \theta(g_1(n) * g_2(n))$$

Establishing order of growth using limits

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

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} (α>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
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

ROAD MAP

- Analysis of algorithms
- Running time functions
- **Mathematical Analysis of Nonrecursive Algorithms**
- Recurrence Relations
  - Exact Solution
    - Forward substitution
    - Backward substitution
    - Methods similar to those used in solving differential equations
  - Asymptotic Solution
    - Master theorem

Time efficiency of nonrecursive algorithms

General Plan for Analysis

- Decide on parameter *n* indicating <u>*input size*</u>
- Identify algorithm's <u>*basic operation*</u>
- Determine <u>*worst*</u>, <u>*average*</u>, and <u>*best*</u> cases for input 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 (see Appendix A)

Properties of Logarithms

1. $\log_a 1 = 0$
2. $\log_a a = 1$
3. $\log_a x^y = y \log_a x$
4. $\log_a xy = \log_a x + \log_a y$
5. $\log_a \dfrac{x}{y} = \log_a x - \log_a y$
6. $a^{\log_b x} = x^{\log_b a}$
7. $\log_a x = \dfrac{\log_b x}{\log_b a} = \log_a b \log_b x$

Important Summation Formulas

1. $\sum_{i=l}^{u} 1 = \underbrace{1 + 1 + \cdots + 1}_{u-l+1 \text{ times}} = u - l + 1$ (l, u are integer limits, $l \le u$); $\sum_{i=1}^{n} 1 = n$

2. $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$

3. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$

4. $\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1}n^{k+1}$

Important Summation Formulas

5. $\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \ (a \neq 1); \quad \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

6. $\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$

7. $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n + \gamma$, where $\gamma \approx 0.5772\ldots$ (Euler's constant)

8. $\sum_{i=1}^{n} \lg i \approx n \lg n$

Sum Manipulation Rules

1. $$\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i$$

2. $$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$$

3. $$\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i, \text{ where } l \le m < u$$

4. $$\sum_{i=l}^{u} (a_i - a_{i-1}) = a_u - a_{l-1}$$

Example : 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 : 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 : 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 $i \leftarrow 0$ **to** $n-1$ **do**
        for $j \leftarrow 0$ **to** $n-1$ **do**
            $C[i, j] \leftarrow 0.0$
            for $k \leftarrow 0$ **to** $n-1$ **do**
                $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
    return C

Example : Counting binary digits

ALGORITHM $Binary(n)$

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
$count \leftarrow 1$
while $n > 1$ **do**
  $count \leftarrow count + 1$
  $n \leftarrow \lfloor n/2 \rfloor$
return $count$

It cannot be investigated the way the previous examples are.