Computer Algorithms

Lecture 2: Fundamentals of the Analysis of Algorithm Efficiency – Ch 2

Dr. Manal Helal, Spring 2014.

http://moodle.manalhelal.com

Lecture Learning Objectives

1. Explain what is meant by "best", "average", and "worst" case behavior of an algorithm
2. In the context of specific algorithms, identify the characteristics of data and/or other conditions or assumptions that lead to different behaviors
3. Determine informally the time and space complexity of simple algorithms.
4. Understand the formal definition of big O
5. Explain the use of big omega, big theta, and little o notation to describe the amount of work done by an algorithm.
6. List and contrast standard complexity classes
7. Use recurrence relations to determine the time complexity of recursively defined algorithms
8. Perform empirical studies to validate hypotheses about runtime stemming from mathematical analysis. Run algorithms on input of various sizes and compare performance

Analysis of algorithms

Issues:

- correctness
- time efficiency
- space efficiency
- optimality

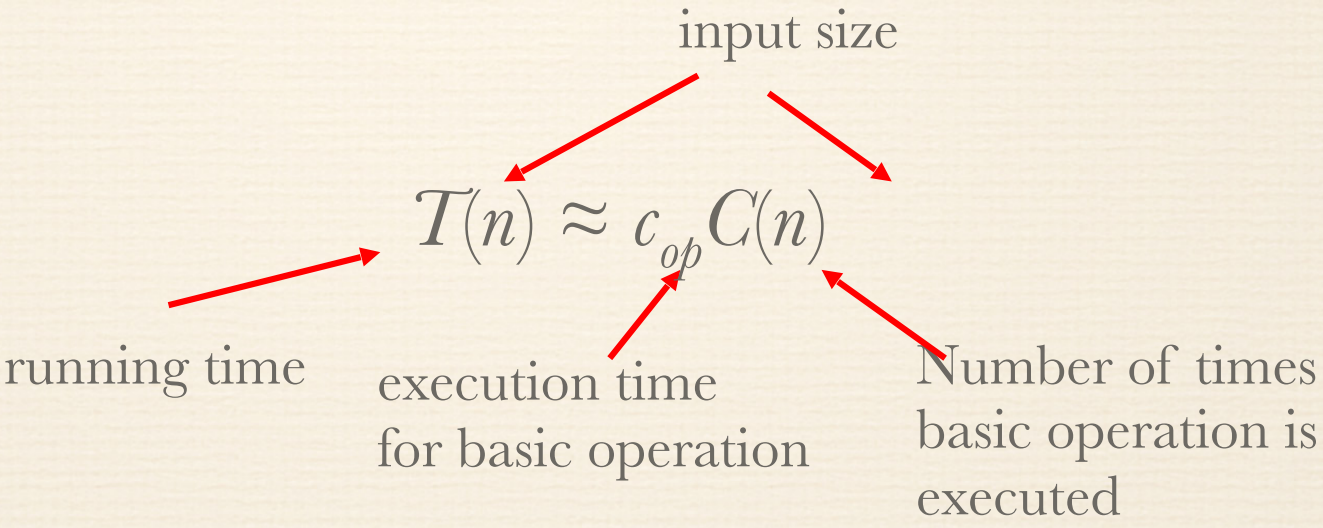
Approaches:

- theoretical analysis
- empirical analysis

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 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

Example

If C(n) = 1 n(n − 1)

What if we double the input size n?

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

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

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

Values of some important functions as $n \to \infty$

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}		

$$\log_a n = \log_a b \log_b n$$

$$\log_2 2n = \log_2 2 + \log_2 n = 1 + \log_2 n$$

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

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 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 $C_{worst}(n) = n.$

Best case $C_{best}(n) = 1$

Average case

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1-p)$$

$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p)$$

$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p).$$

If p = 1, $C_{avg} = (n+1)/2$

Types of Formulas For Basic Operation's Count

Exact formula

e.g., $C(n) = n(n\text{-}1)/2$

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

Algorithm Execution-Time Analysis

Sum the natural numbers 1 to n.

Arithmetic series.
We know that SUM = 1+2+...+n = n(n+1)/2.

A1

```
Function SUM(n: ℕ) : ℕ
    result := n(n+1)/2
    return result
```

A2

```
Function SUM(n: ℕ) : ℕ
    result := 0
    for i:=1 to n do
        result := result + i
    end for
    return result
```

Which algorithm is faster A1 or A2?

Algorithm Execution-Time Analysis

Algorithm A1

A1 computes the result in a **constant** number of steps at runtime.

What is the execution time T(A1) for A1?

$T(A1) = \beta_1$ (β_1 is some constant)

T(A1) does not depend on the input.

If input is 1, 10, 100, etc. T(A1) will not change (significantly).

Algorithm A2

A2 computes the result in a **variable** number of steps at runtime depending on input size.

What is the time T(A2) for A2?

$T(A2) = \alpha_2 n + \beta_2$ (α_2, β_2 are some constants)

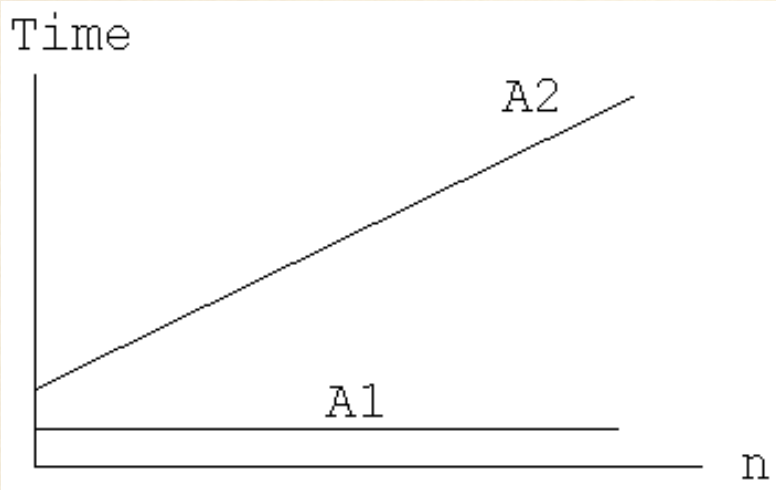
T(A2) depends on the input size

If input is 1, 10, 100, etc. T(A2) will change.

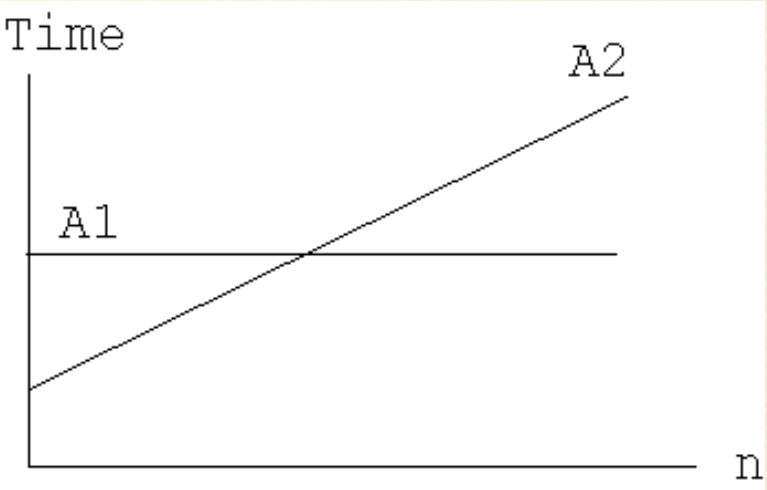
Algorithm Execution-Time Analysis

Which of A1 and A2 is faster?

- $T(A1) = \beta_1$
- $T(A2) = \alpha_2 n + \beta_2$



When is this the case?



When is this the case?

Algorithm Execution-Time Analysis

We say that A1 has a **constant growth**.

Its execution time does not depend on the input value/size.

$T(A1) = \beta_1$

We say that A2 has a **linear growth** in terms of the input.

Its execution time grows as a linear function in terms of the input value/size.

$T(A2) = \alpha_2 n + \beta_2$

A1 is more efficient than A2.

When input is small, A2 might be more efficient.

Analysis of algorithms when input is small is generally not interesting, as they spend most of the time in these cases in **start-up** code.

When input is large, A1 is more efficient.

We are interested in the cases when input is large, also called as the **difficult instances**.

So A1 is more efficient than A2.

Exercise

2.1.1. For each of the following algorithms, indicate (i) a natural size metric for its inputs, (ii) its basic operation, and (iii) whether the basic operation count can be different for inputs of the same size:

a. Computing the sum of n numbers

b. Computing n!

c. Finding the largest element in a list of n numbers

d. Euclid's algorithm

e. Sieve of Eratosthenes

f. Pen-and-pencil algorithm for multiplying two n-digit decimal integers

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?

Asymptotic Analysis – Big Oh

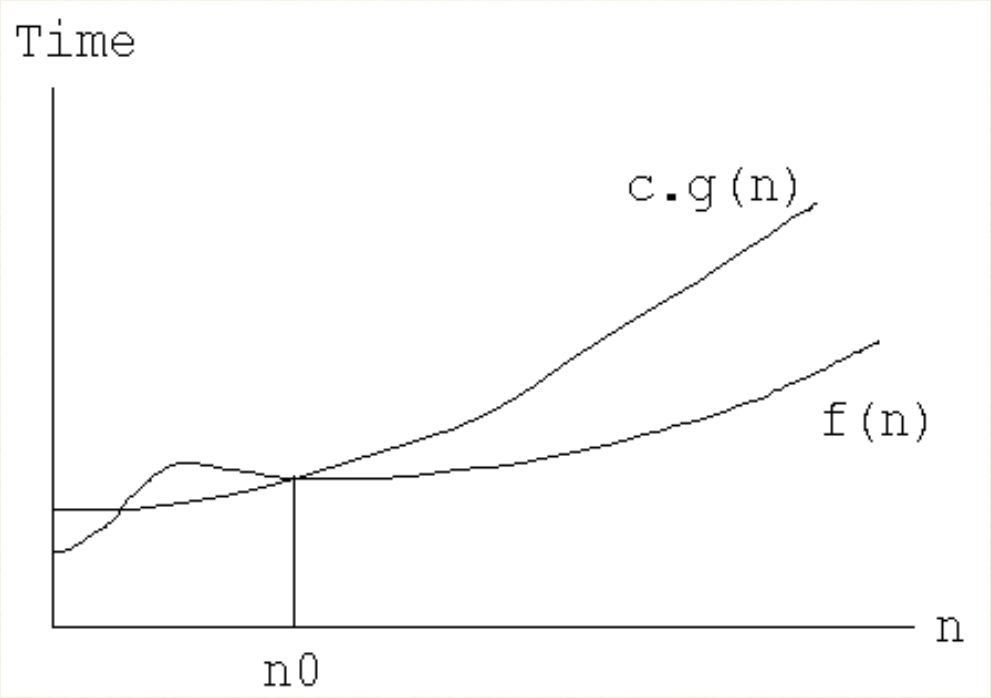
Assume algorithms A1 and A2 with growth functions f and g respectively.

We say that “f grows no faster than g in the limit” if:

$O(g(n)) = \{f(n) | \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}^+ \cdot (\forall n \in \mathbb{N} \cdot n \geq n_0 \Rightarrow f(n) \leq c.g(n))\}$

Written as: $f(n) \in O(g(n))$
Read as: f is “big Oh of” g
Also: f is “asymptotically dominated by” g
Also: g is an upper bound on f

Asymptotic Analysis – Big Oh



Example – Big Oh

Let $f(n)=3n^2+4n+5$, let $g(n)=n^2$

Is $f(n)=O(g(n))$?

i.e., is $3n^2+4n+5 = O(n^2)$

Let's prove it.

- $f(n)=O(g(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}^+ \cdot (\forall n \in \mathbb{N} \cdot n \geq n_0 \Rightarrow f(n) \leq c.g(n))$

- Can we find c, n_0 such that the above equivalence is correct?
  - We know that: $\forall n \geq 1: 3n^2+4n+5 \leq 3n^2+4n^2+5n^2$
  - i.e., $\forall n \geq 1: 3n^2+4n+5 \leq 12n^2$
  - i.e., $n_0=1$, $c=12$

More examples:

$3n^2+4n+5 = O(n^2)$

$3n^2+4n+5 = O(n^3)$

$3n^2+4n+5 \neq O(n)$

Asymptotic Analysis – Big Omega

Big Oh notation

- f(n) = O(g(n))
- f(n) is big Oh of g(n)
- f(n) grows no faster than g(n)
- $f(n)= O(g(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}^+ \cdot (\forall n \in \mathbb{N} \cdot n \geq n_0 \Rightarrow f(n) \leq c.g(n))$

Big Omega notation

- f(n) = Ω(g(n))
- f(n) is big Omega of g(n)
- f(n) grows no slower than g(n)
- $\Omega(g(n)) = \{f(n) | \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}^+ \cdot (\forall n \in \mathbb{N} \cdot n \geq n_0 \Rightarrow f(n) \geq c.g(n))\}$

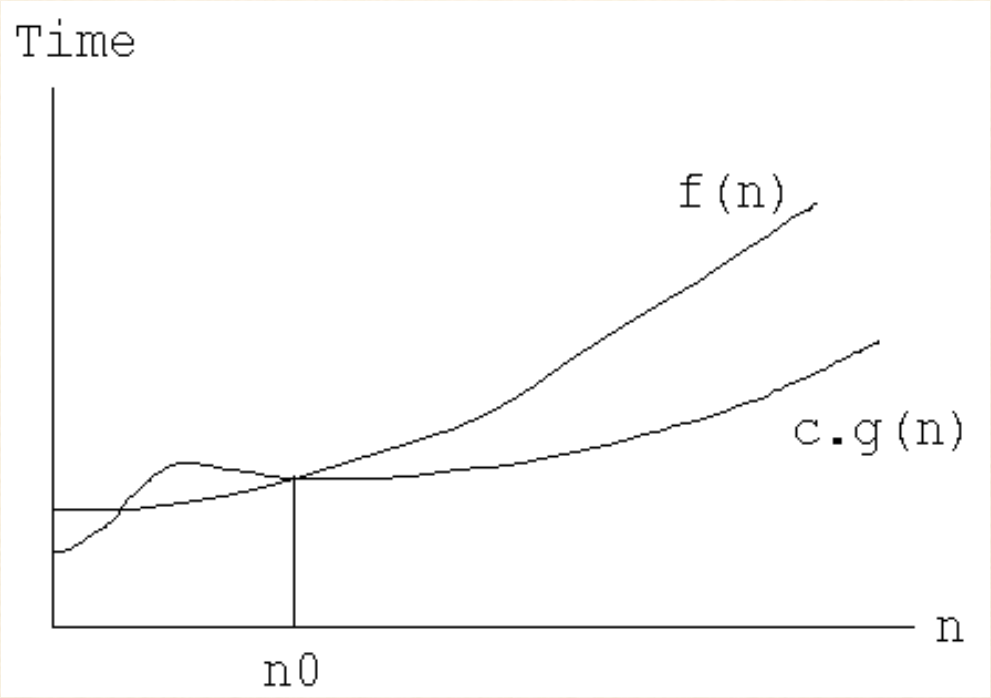
Relation

- f(n) = O(g(n)) ⇔ g(n) = Ω(f(n))

More examples:

- $3n^2+4n+5 = \Omega(n^2)$
- $3n^2+4n+5 \neq \Omega(n^3)$
- $3n^2+4n+5 = \Omega(n)$

Asymptotic Analysis – Big Omega



Asymptotic Analysis – Big Theta

Big Theta notation

f(n) = Θ(g(n))
f(n) is big Theta of f(n)
f(n) grows as the same rate as g(n)

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

$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

Asymptotic Analysis – Big Theta

Big Theta notation

$f(n) = \Theta(g(n))$
$f(n)$ is big Theta of $g(n)$
$f(n)$ grows as the same rate as $g(n)$

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

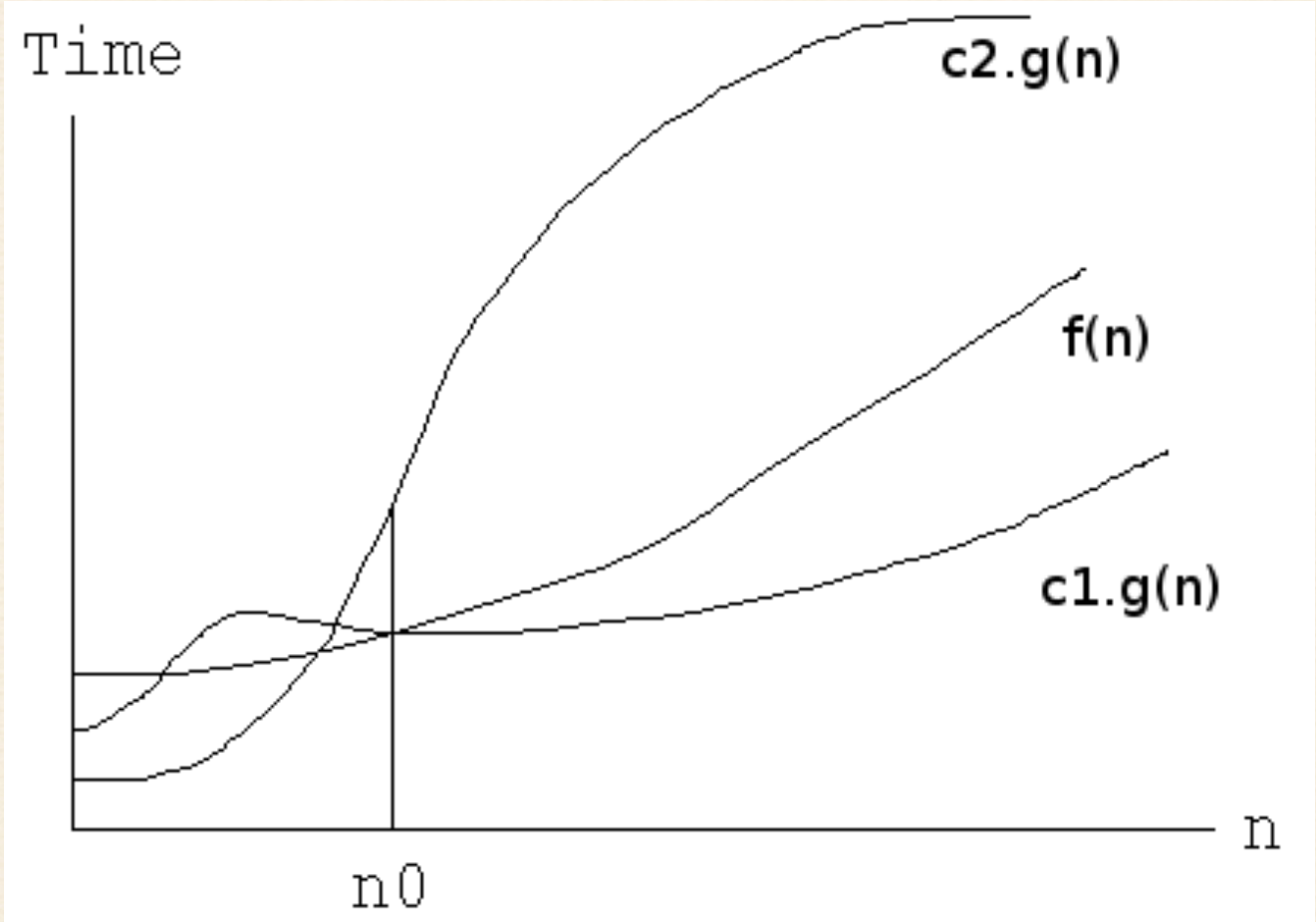
$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

$$f(n)= \Theta(g(n)) \Leftrightarrow \exists c_1 \in \mathbb{R}^+, \exists c_2 \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}^+ \cdot (\forall n \in \mathbb{N} \cdot n \geq n_0 \Rightarrow c_1.g(n) \leq f(n) \leq c_2.g(n))$$

More examples:

$3n^2+4n+5 = \Theta(n^2)$
$3n^2+4n+5 \neq \Theta(n^3)$
$3n^2+4n+5 \neq \Theta(n)$

Asymptotic Analysis – Big Theta



Asymptotic Analysis – Little Oh

Little Oh notation

- $f(n) = o(g(n))$
- f(n) is little Oh of g(n)
- f(n) grows strictly slower than g(n)

$f(n)= o(g(n)) \Leftrightarrow (f(n) = O(g(n)) \wedge f(n) \neq \Theta(g(n)))$

$f(n)=o(g(n)) \Leftrightarrow \forall c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}^+ \cdot (\forall n \in \mathbb{N} \cdot n \geq n_0 \Rightarrow f(n)<c.g(n))$

More examples:

- $3n^2+4n+5 \neq o(n^2)$
- $3n^2+4n+5 = o(n^3)$
- $3n^2+4n+5 \neq o(n)$

Asymptotic Analysis – Little Omega

Little Omega notation

- $f(n) = \omega(g(n))$
- f(n) is little Omega of g(n)
- f(n) grows strictly faster than g(n)

$f(n)= \omega(g(n)) \Leftrightarrow (f(n) = \Omega(g(n)) \wedge f(n) \neq \Theta(g(n)))$

$f(n)= \omega(g(n)) \Leftrightarrow \forall c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}^+ \cdot (\forall n \in \mathbb{N} \cdot n \geq n_0 \Rightarrow f(n) > c.g(n))$

More examples:

- $3n^2+4n+5 \neq \omega(n^2)$
- $3n^2+4n+5 \neq \omega(n^3)$
- $3n^2+4n+5 = \omega(n)$

Asymptotic Analysis – Summary

We have seen notation that we can use to describe the growth of a function.

- Big Oh
- Big Omega
- Big Theta
- Little Oh
- Little Omega

Instead of dealing with complex functions e.g., we just look at the higher-order terms and drop constants.

For example:

$5n^4 + 2n^3 + 3n^2 + 4$

Asymptotic Analysis – Summary

We have seen notation that we can use to describe the growth of a function.

- Big Oh
- Big Omega
- Big Theta
- Little Oh
- Little Omega

Instead of dealing with complex functions e.g., we just look at the higher-order terms and drop constants.

For example:

$5n^4 + 2n^3 + 3n^2 + 4 = \Theta(n^4)$.

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

$$\lim_{n\to\infty} T(n)/g(n) =$$

- **0** order of growth of ***T(n)*** < order of growth of ***g(n)***
- ***c* > 0** order of growth of ***T(n)*** = order of growth of ***g(n)***
- **∞** order of growth of ***T(n)*** > order of growth of ***g(n)***

$t(n) \in O(g(n))$
$t(n) \in \Omega_{(g(n))}$
$t(n) \in \Theta_{(g(n))}$

Example:

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

$$\frac{1}{2}n(n-1) \in \Theta(n^2)$$

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:

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$$

$$\log_2 n \in o(\sqrt{n})$$

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

Example:

$$\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n} = \lim_{n\to\infty} \sqrt{2\pi n}\frac{n^n}{2^n e^n} = \lim_{n\to\infty} \sqrt{2\pi n}\left(\frac{n}{2e}\right)^n = \infty$$

$$n! \in \Omega(2^n)$$

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

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
∞	Incomputable

Time Efficiency of Non-Recursive Algorithms

General Plan for Analysis:

1. Decide on parameter *n* indicating *input size*

2. Identify algorithm's *basic operation*

3. Determine *worst*, *average*, and *best* cases for input of size *n*

1. Set up a sum for the number of times the basic operation is executed

2. Simplify the sum using standard formulas and rules (see Appendix A)

Useful summation formulas and rules

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

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

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

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

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

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

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i$

$\Sigma c a_i = c \Sigma a_i$

$\Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

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 $maxval$

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Example 2: Element Uniqueness

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

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

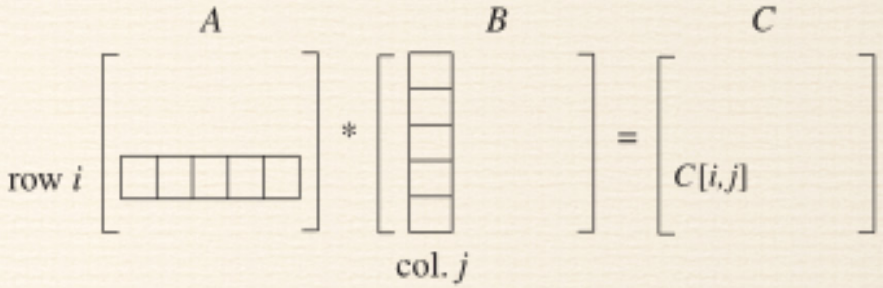
$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).$$

Example 3: Matrix multiplication

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

//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two $n \times n$ 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



$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3$$

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3$$

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
$count \leftarrow 1$
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.

$$\mathbf{O}(\log_2 n)$$

Plan for Analysis of Recursive Algorithms

- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

Example 1: Recursive evaluation of $n!$

Definition: $n! = 1 \cdot 2 \cdot \ldots \cdot (n\text{-}1) \cdot n$ for $n \geq 1$ and $0! = 1$

Recursive definition of $n!$: $F(n) = F(n\text{-}1) \cdot n$ for $n \geq 1$ and $F(0) = 1$

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

Size:
Basic operation:
Recurrence relation:

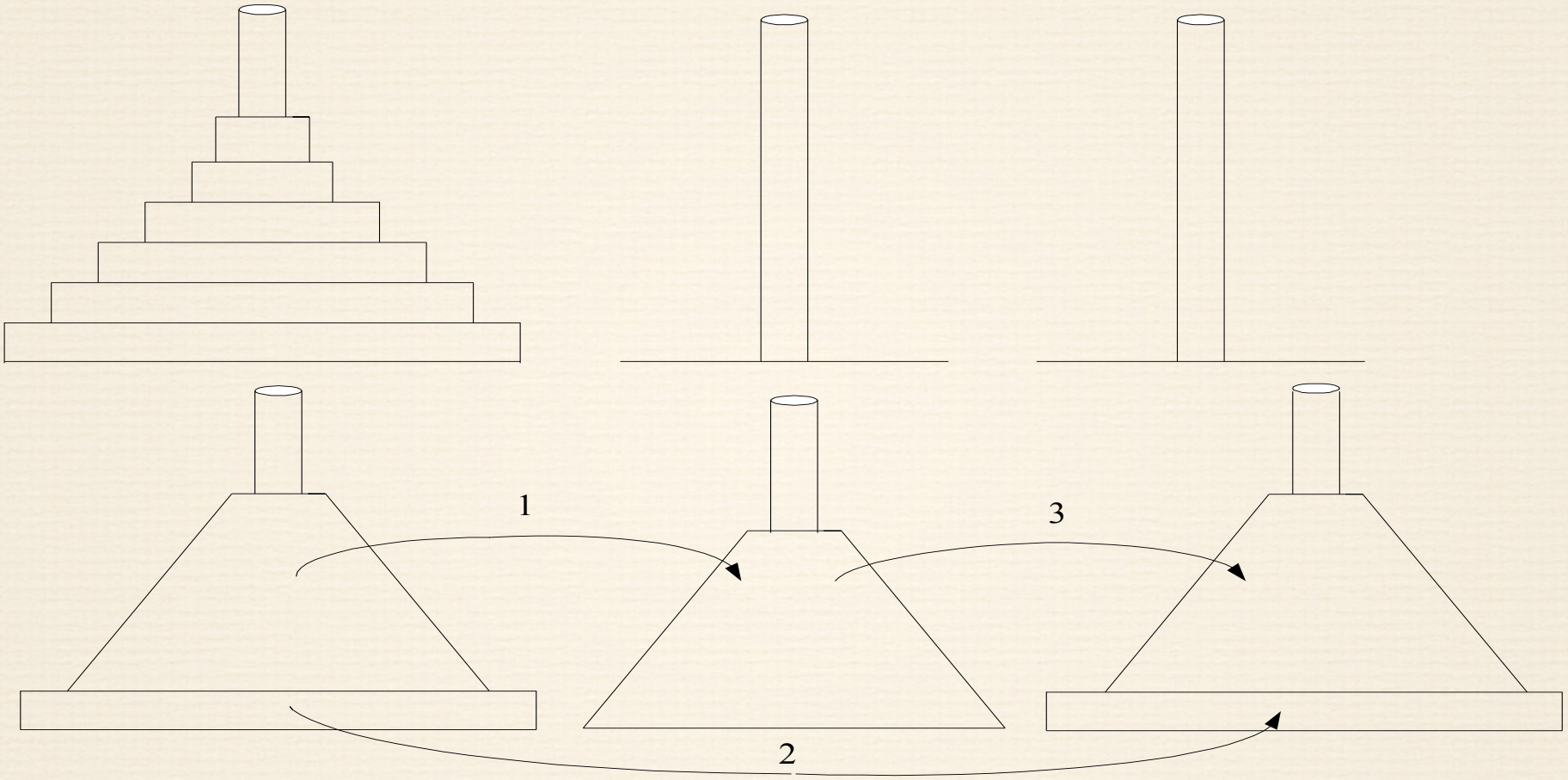
Solving the recurrence for M(n)

$M(n) = M(n\text{-}1) + 1, \; M(0) = 0$

$$
\begin{aligned}
M(n) &= M(n-1) + 1 && \text{substitute } M(n-1) = M(n-2) + 1 \\
&= [M(n-2) + 1] + 1 = M(n-2) + 2 && \text{substitute } M(n-2) = M(n-3) + 1 \\
&= [M(n-3) + 1] + 2 = M(n-3) + 3.
\end{aligned}
$$

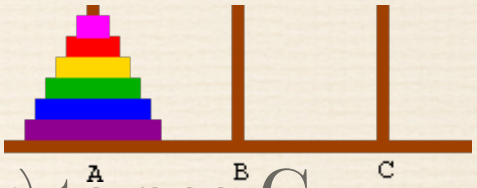
$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$

Example 2: The Tower of Hanoi Puzzle



Recurrence for number of moves:

Tower of Hanoi Solution



Step 1: Move disks 4 and smaller from peg A (*source*) to peg C (*spare*), using peg B (*dest*) as a spare. How do we do this? By recursively using the same procedure. After finishing this, we'll have all the disks smaller than disk 4 on peg C.

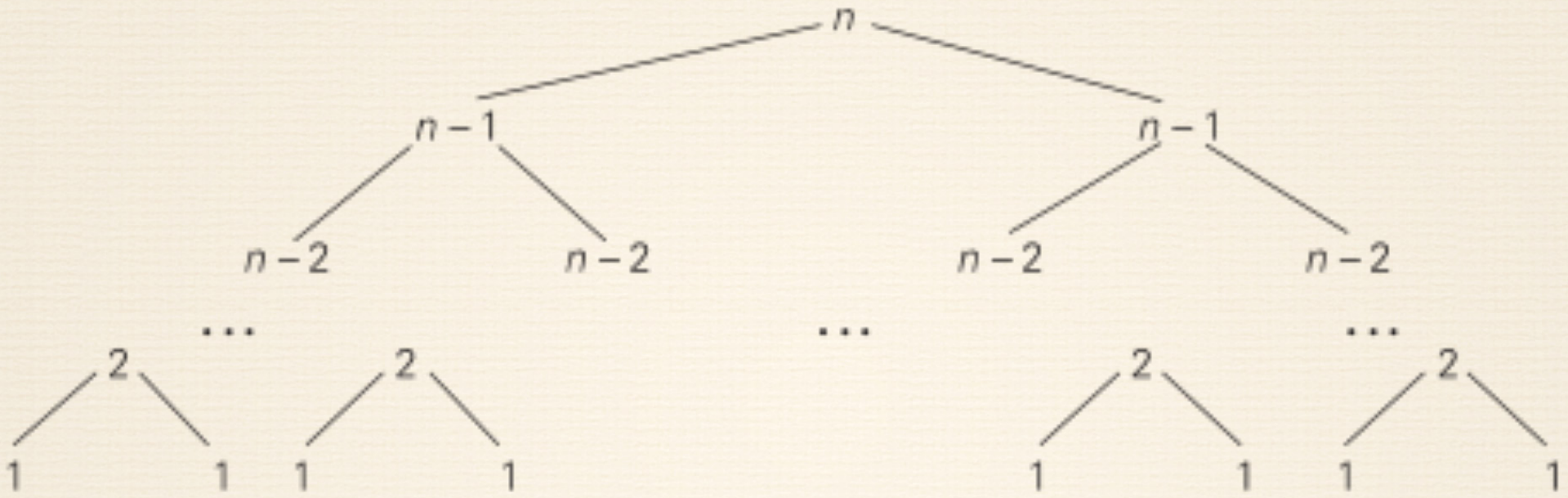
Step 2: Now, with all the smaller disks on the spare peg, we can move disk 5 from peg A (source) to peg B (dest).

Step 3: Finally, we want disks 4 and smaller moved from peg C (spare) to peg B (dest). We do this recursively using the same procedure again. After we finish, we'll have disks 5 and smaller all on dest.

Tower of Hanoi Recursive Algorithm

```
FUNCTION MoveTower(disk, source, dest, spare):
IF disk == 0, THEN:
    move disk from source to dest
ELSE:
    MoveTower(disk - 1, source, spare, dest)   // Step 1 above
    move disk from source to dest              // Step 2 above
    MoveTower(disk - 1, spare, dest, source) // Step 3 above
END IF
```

Tree of Calls for the Tower of Hanoi Puzzle



Solving Recurrence for Number of Moves

$$M(n) = M(n-1) + 1 + M(n-1) \quad \text{for } n > 1.$$

$$\mathrm{M}(n) = 2\mathrm{M}(n\text{-}1) + 1, \ \mathrm{M}(1) = 1$$

$$M(n) = 2M(n-1) + 1 \qquad \text{sub. } M(n-1) = 2M(n-2) + 1$$

$$= 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1 \quad \text{sub. } M(n-2) = 2M(n-3) + 1$$

$$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1.$$

$$2^4 M(n-4) + 2^3 + 2^2 + 2 + 1,$$

$$\cdots$$

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n-i) + 2^i - 1.$$

$$\cdots$$

$$M(n) = 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1$$

$$= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$$