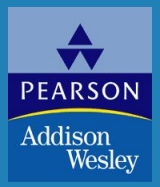
Chapter 2

Fundamentals of the Analysis of Algorithm Efficiency



Analysis of algorithms

- **Issues:**
  - **correctness**
  - **time efficiency**
  - **space efficiency**
  - **optimality**
- **Approaches:**
  - **theoretical analysis**
  - **empirical analysis**

Analysis of algorithms

The efficiency of an algorithm can be in terms of time and space. The algorithm efficiency can be analyzed by the following ways.

a. Analysis Framework.

b. Asymptotic Notations and its properties.

c. Mathematical analysis for Recursive algorithms.

d. Mathematical analysis for Non-recursive algorithms.

Analysis of algorithms

Analysis Framework

There are two kinds of efficiencies to analyze the efficiency of any algorithm. They are:

- **Time efficiency, indicating how fast the algorithm runs, and**
- **Space efficiency, indicating how much extra memory it uses.**

The algorithm analysis framework consists of the following:

- **Measuring an Input's Size.**
- **Units for Measuring Running Time.**
- **Orders of Growth .**
- **Worst-Case, Best-Case, and Average-Case Efficiencies**

Measuring an Input's Size

An algorithm's efficiency is defined as a function of some parameter n indicating the algorithm's input size. In most cases, selecting such a parameter is quite straightforward.

For example, it will be the size of the list for problems of sorting, searching.

- For the problem of evaluating a polynomial

p(x) = $a_n x^n$ + . . . + a0 of degree n, the size of the parameter will be the polynomial's degree or the number of its coefficients, which is larger by 1 than its degree.

- In computing the product of two n × n matrices, the choice of a parameter indicating an input size does matter.

Measuring an Input's Size

- Consider a spell-checking algorithm. If the algorithm examines individual characters of its input, then the size is measured by the number of characters.
- In measuring input size for algorithms solving problems such as checking primality of a positive integer n. the input is just one number.
- The input size by the number b of bits in the n's binary representation is b=(log2 n)+1.

Units for Measuring Running Time

Some standard unit of time measurement such as a second, or millisecond, and so on can be used to measure the running time of a program after implementing the algorithm.

But has some drawbacks:

- Dependence on the speed of a particular computer.
- Dependence on the quality of a program implementing the algorithm.
- The compiler used in generating the machine code.
- The difficulty of clocking the actual running time of the program.

So, we need metric to measure an algorithm's efficiency that does not depend on these extraneous factors.

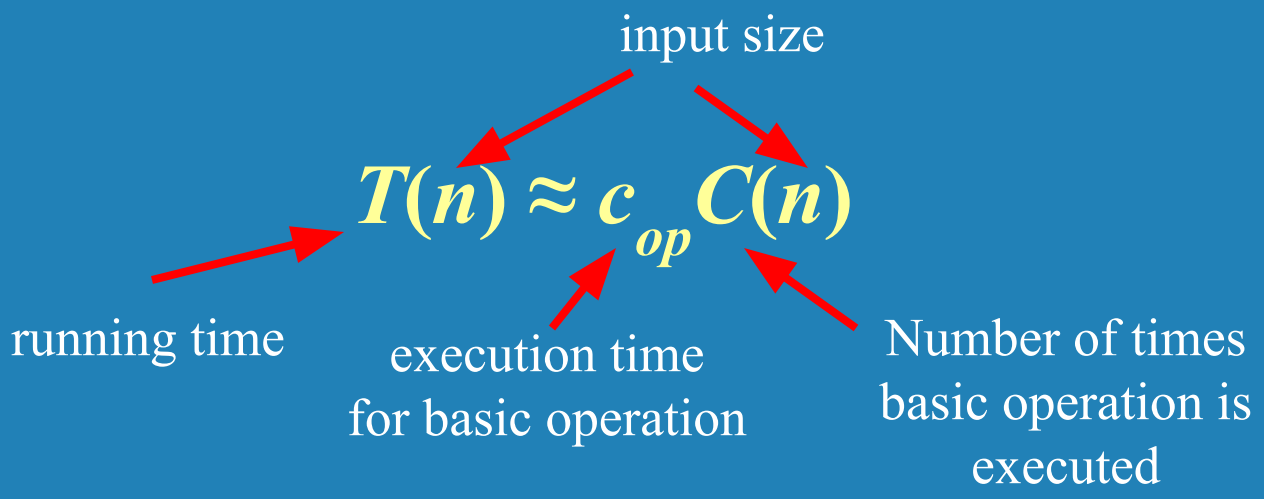
Units for Measuring Running Time

- One possible approach is to count the number of times each of the algorithm's operations is executed.
- The most important operation (+, -, *, /) of the algorithm, called the "*basic operation*".
- Computing the number of times the basic operation is executed is easy.
- The total running time is determined by basic operations count.

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

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

Order of growth

- A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.
- For example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid's algorithm is compared to the other algorithms, the difference in algorithm efficiencies becomes clear for larger numbers only.
- For large values of n, it is the function's order of growth that counts just like the Table 2.1, which contains values of a few functions particularly important for analysis of algorithms.
- 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 $n \to \infty$

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**

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
```

- **Worst case**
- **Best case**
- **Average case**

Example: Sequential search

- **Worst case:**
  - The worst-case efficiency of an algorithm is its efficiency for the worst case input of size n.
  - The algorithm runs the longest among all possible inputs of that size.
  - For the input of size n, the running time is Cworst(n) = n.

- **Best case:**
  - The best-case efficiency of an algorithm is its efficiency for the best case input of size n.
  - The algorithm runs the fastest among all possible inputs of that size n.
  - In sequential search, If we search a first element in list of size n. (i.e. first element equal to a search key), then the running time is Cbest(n) = 1

Example: Sequential search

- Average case
  - The Average case efficiency lies between best case and worst case.
  - To analyze the algorithm's average case efficiency, we must make some assumptions about possible inputs of size n.
  - The standard assumptions are that
    - The probability of a successful search is equal to p ($0 \le p \le 1$) and
    - The probability of the first match occurring in the i^{th} position of the list is the same for every i

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

- Yet another type of efficiency is called amortized efficiency. It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure.

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

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 $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$

Big-oh

(i) O - Big oh notation

A function t(*n*) is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

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

Where $t(n)$ and $g(n)$ are nonnegative functions defined on the set of natural numbers.

O = Asymptotic upper bound = Useful for worst case analysis = Loose bound

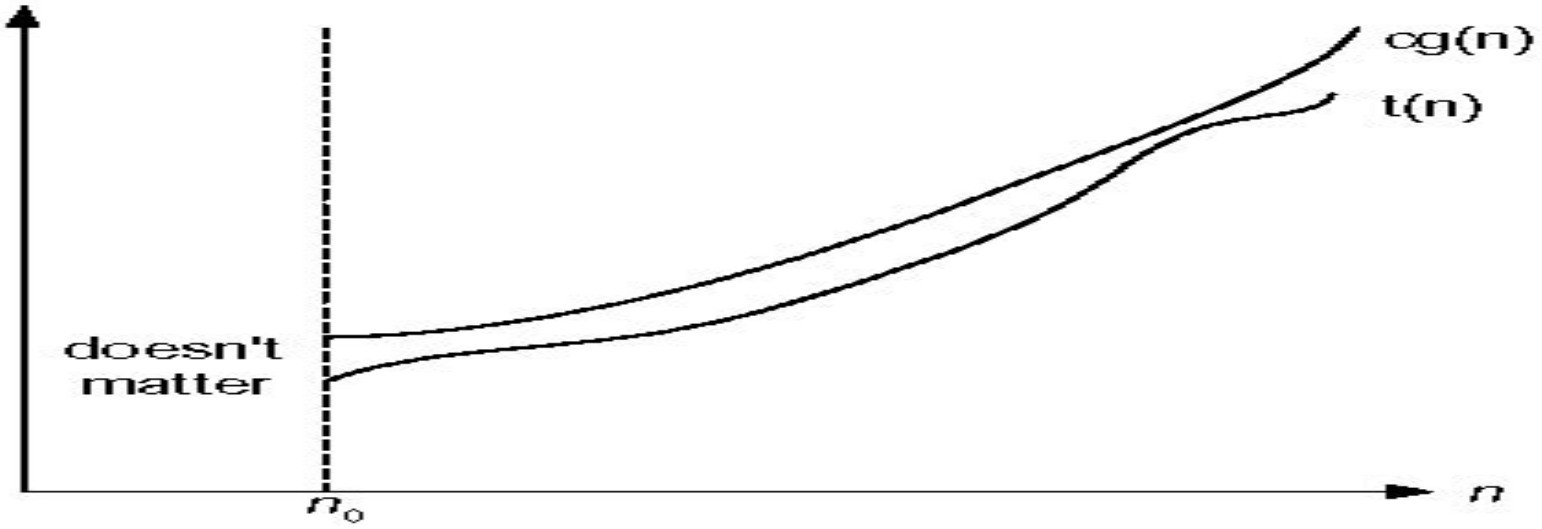


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega

(ii) Ω - Big omega notation

A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

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

Where $t(n)$ and $g(n)$ are nonnegative functions defined on the set of natural numbers.

Ω = Asymptotic lower bound = Useful for best case analysis = Loose bound

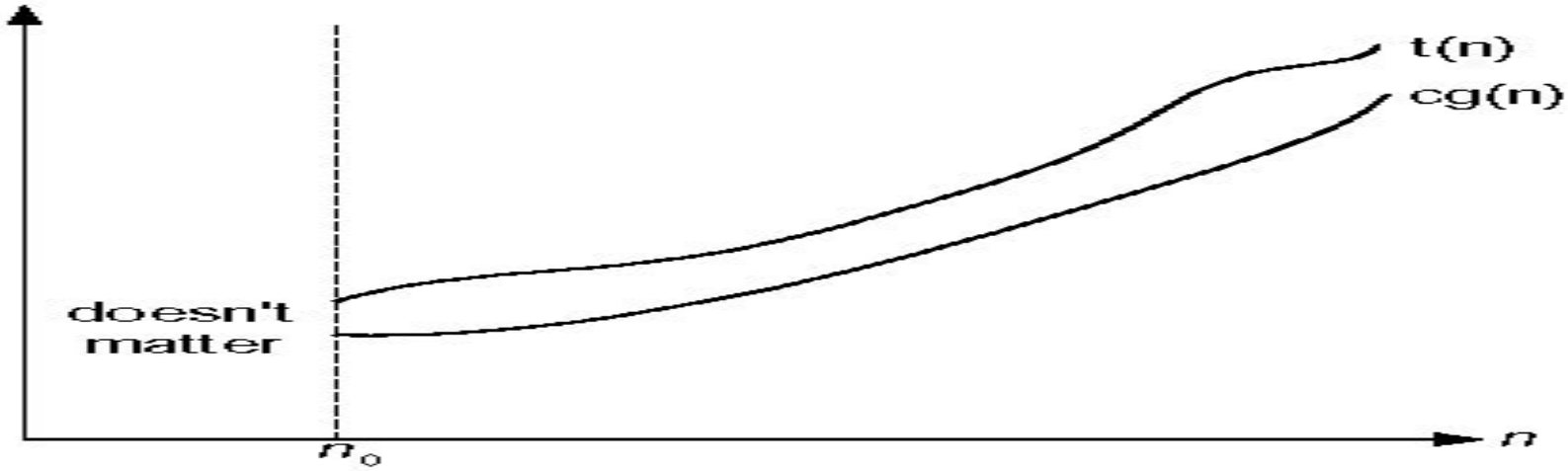


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

(iii) Θ - Big theta notation

A function t(n) is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if t(n) is bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0.$$

Where *t(n)* and *g(n)* are nonnegative functions defined on the set of natural numbers.

Θ = Asymptotic tight bound = Useful for average case analysis

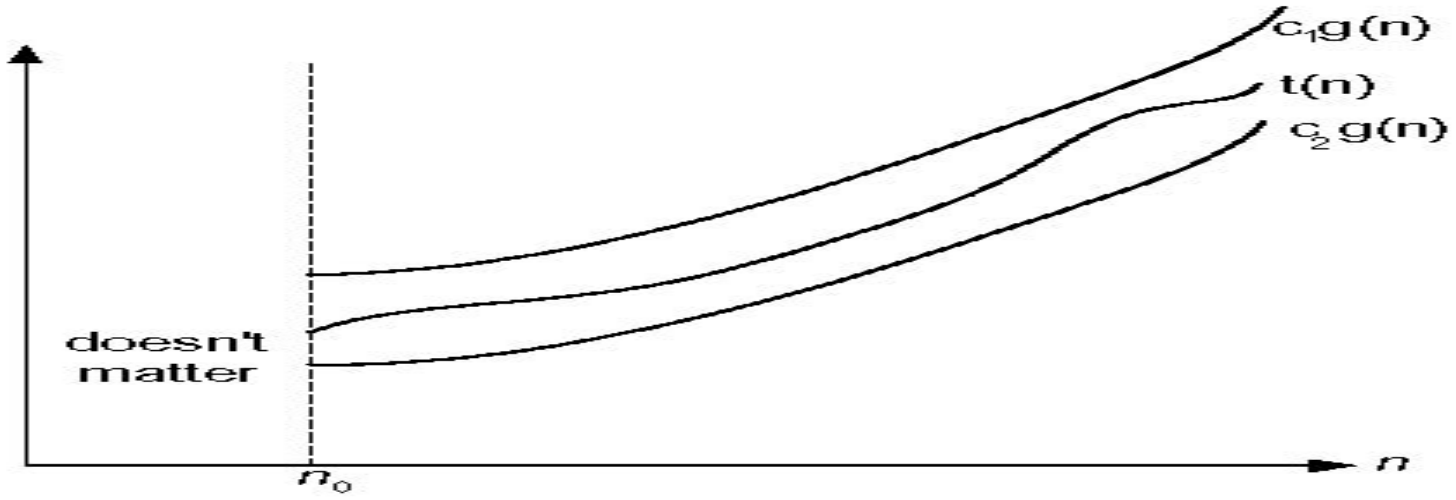


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

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

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

- 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 $g(n)$

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

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

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

Example: $\log n$ vs. 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
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 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.

Useful summation formulas and rules

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

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

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

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

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

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

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

Useful summation formulas and rules

$$\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \text{ are some lower and upper integer limits,} \qquad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2). \qquad \textbf{(S2)}$$

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 ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

Example 1: Maximum element

- The measure of an input's size here is the number of elements in the array, i.e., n.
- There are two operations in the for loop's body:
  - The comparison A[i]> maxval and
  - The assignment maxval←A[i].
- The comparison operation is considered as the algorithm's basic operation, because the comparison is executed on each repetition of the loop and not the assignment.

Example 1: Maximum element

- The number of comparisons will be the same for all arrays of size n; therefore, there is no need to distinguish among the worst, average, and best cases here.
- Let C(n) denotes the number of times this comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n − 1, inclusive. Therefore, the sum for C(n) is calculated as follows:

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

i.e., Sum up 1 in repeated n-1 times

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

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 ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

Example 2: Element uniqueness problem

- The natural measure of the input's size here is again n (the number of elements in the array).
- Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation.
- The number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only.

Example 2: Element uniqueness problem

- One comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable j between its limits i + 1 and n − 1; this is repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and n − 2.

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

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

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

We also could have computed the sum $\sum_{i=0}^{n-2} (n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2},$$

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 $i \leftarrow 0$ **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 3: Matrix multiplication

- An input's size is matrix order n.
- There are two arithmetical operations (multiplication and addition) in the innermost loop. But we consider multiplication as the basic operation.
- Let us set up a sum for the total number of multiplications M(n) executed by the algorithm. Since this count depends only on the size of the input matrices, we do not have to investigate the worst-case, average-case, and best-case efficiencies separately.
- There is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable k ranging from the lower bound 0 to the upper bound $n - 1$.

Example 3: Matrix multiplication

- **Therefore, the number of multiplications made for every pair of specific values of variables i and j is**

$$\sum_{k=0}^{n-1} 1$$

The total number of multiplications M(n) is expressed by the following triple sum:

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

Now, we can compute this sum by using formula (S1) and rule (R1)

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

The running time of the algorithm on a particular machine m, we can do it by the product

$$T(n) \approx c_m M(n) = c_m n^3,$$

If we consider, time spent on the additions too, then the total time on the machine is

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

Example 5: 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
```

It cannot be investigated the way the previous examples are.

Example 5: Counting binary digits

- An input's size is n.
- The loop variable takes on only a few values between its lower and upper limits.
- Since the value of n is about halved on each repetition of the loop, the answer should be about $\log_2 n$.
- The exact formula for the number of times.
- The comparison n > 1 will be executed is actually $[\log_2 n] + 1$

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

Recursive definition of $n!$: $F(n) = F(n\text{-}1) * n$ 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*)

- For simplicity, we consider n itself as an indicator of this algorithm's input size. i.e. 1.
- The basic operation of the algorithm is multiplication, whose number of executions we denote M(n). Since the function F(n) is computed according to the formula.

$$F(n) = F(n-1)*n \quad \text{for } n > 0$$

- The number of multiplications M(n) needed to compute it must satisfy the equality

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute} \\ F(n-1)}} + \underbrace{1}_{\substack{\text{to multiply} \\ F(n-1) \text{ by } n}} \quad \text{for } n > 0.$$

- M(n − 1) multiplications are spent to compute F(n − 1), and one more multiplication is needed to multiply the result by n.

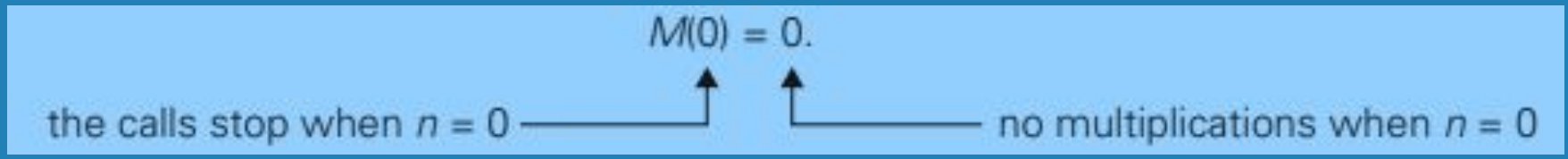
Recurrence Relations

- The last equation defines the sequence M(n) that we need to find. This equation defines M(n) not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely n − 1. Such equations are called recurrence relations or recurrences.
- Solve the recurrence relation $M(n) = M(n-1) + 1$, i.e., to find an explicit formula for M(n) in terms of n only.
- To determine a solution uniquely, we need an *initial condition* that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if $n = 0$ **return** 1.

Recurrence Relations

- Two things to understand.
- First, since the calls stop when n = 0, the smallest value of n for which this algorithm is executed and hence M(n) defined is 0.
- Second, by inspecting the pseudocode's exiting line, we can see that when n = 0, the algorithm performs no multiplications.



- Thus, the recurrence relation and initial condition for the algorithm's number of multiplications:

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

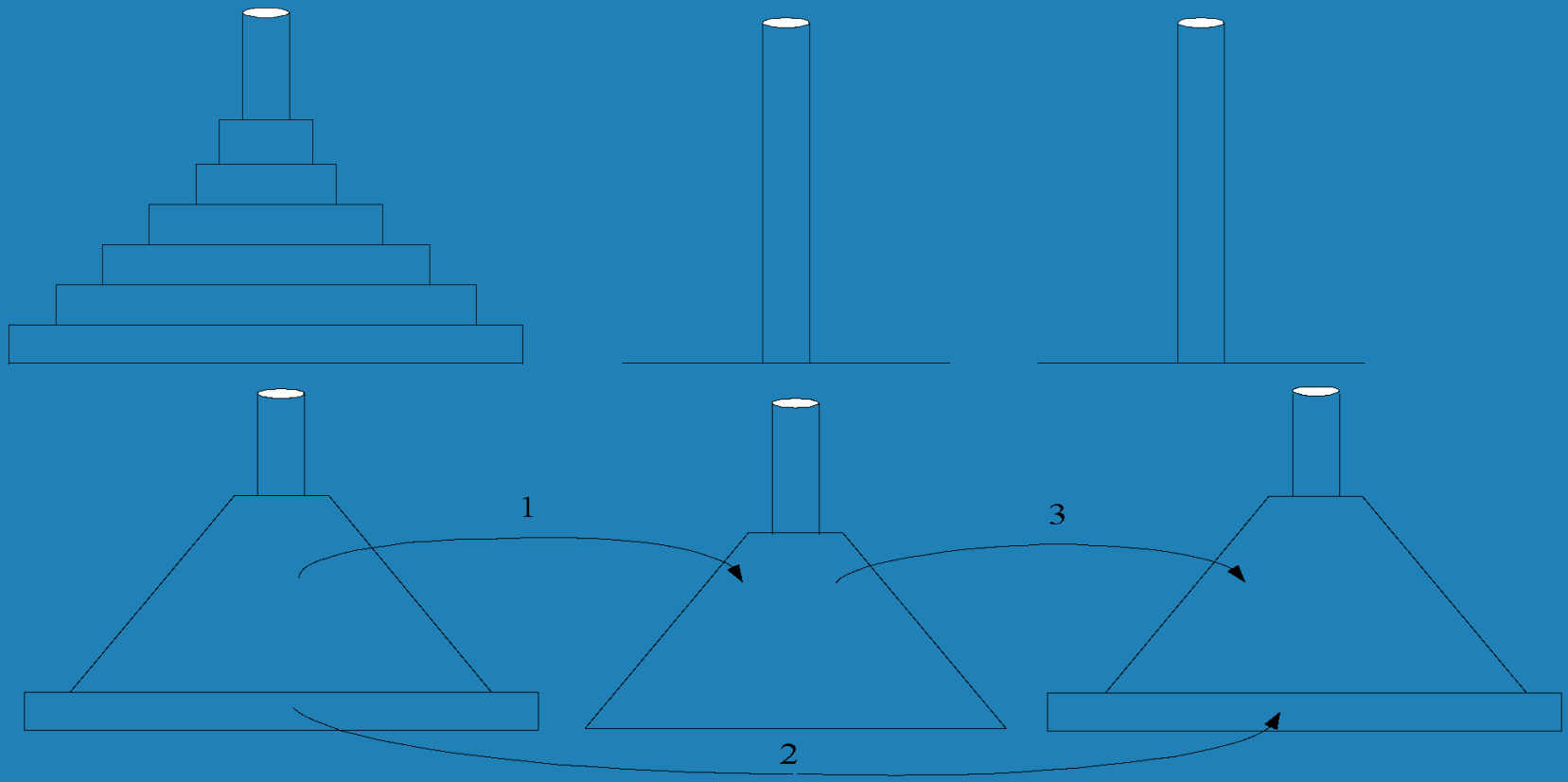
Method of Backward Substitutions

Method of backward substitutions

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

Therefore $M(n)=n$

Example 2: The Tower of Hanoi Puzzle



Recurrence for number of moves:

Solving recurrence for number of moves

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

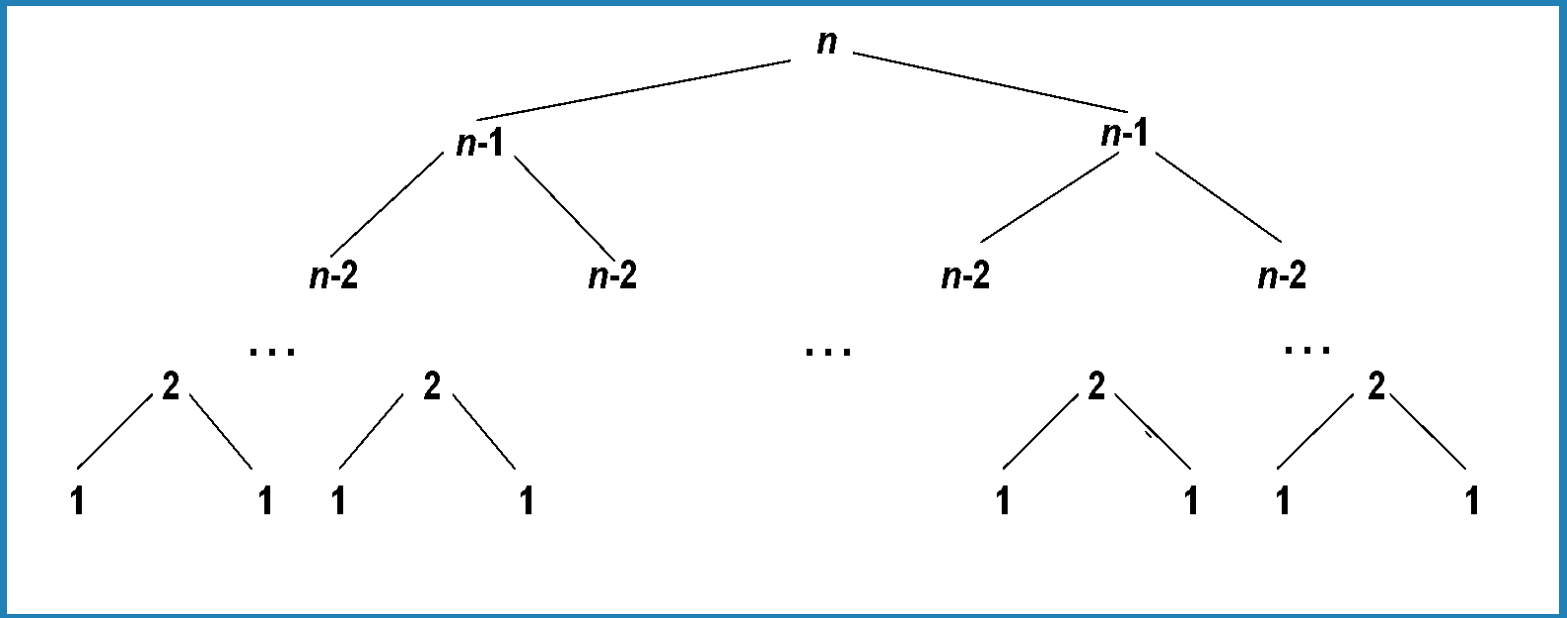
We solve this recurrence by the same method of backward substitutions:

$$\begin{aligned}
M(n) &= 2M(n-1) + 1 && \text{sub. } M(n-1) = 2M(n-2) + 1\\
&= 2[2M(n-2) + 1] + 1\\
&= 2^2M(n-2) + 2 + 1 && \text{sub. } M(n-2) = 2M(n-3) + 1\\
&= 2^2[2M(n-3) + 1] + 2 + 1\\
&= 2^3M(n-3) + 2^2 + 2 + 1 && \text{sub. } M(n-3) = 2M(n-4) + 1\\
&= 2^4M(n-4) + 2^3 + 2^2 + 2 + 1\\
&\ldots\\
&= 2^iM(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 2 + 1 = 2^iM(n-i) + 2^i - 1.\\
&\ldots
\end{aligned}$$

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$,

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

Tree of calls for the Tower of Hanoi Puzzle



Example 3: Counting #bits

ALGORITHM *BinRec*(n)

//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

if $n = 1$ **return** 1

else return *BinRec*$(\lfloor n/2 \rfloor) + 1$

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1.$$
$$A(1) = 0.$$

Example 3: Counting #bits

The standard approach to solving such a recurrence is to solve it only for $n = 2^k$

$$A(2^k) = A(2^{k-1}) + 1 \text{ for } k > 0,$$
$$A(2^0) = 0.$$

backward substitutions

$A(2^k) = A(2^{k-1}) + 1$ substitute $A(2^{k-1}) = A(2^{k-2}) + 1$

$= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2$ substitute $A(2^{k-2}) = A(2^{k-3}) + 1$

$= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3$ . . .

. . .

$= A(2^{k-i}) + i$

. . .

$= A(2^{k-k}) + k.$

Thus, we end up with $A(2^k) = A(1) + k = k$, or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$A(n) = \log_2 n \in \Theta(\log_2 n)$.