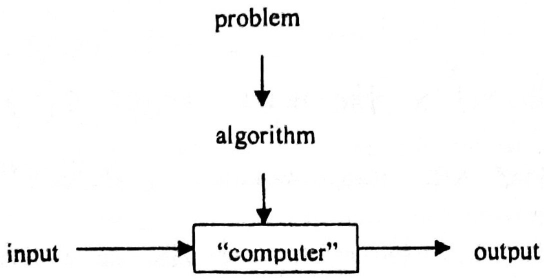
Chapter 1. The Role of Algorithms in Computing

- An algorithm is a sequence of unambiguous instructions for solving a problem ,i.e., for obtaining a required output for any legitimate input in a finite amount of time.

- More formally, an algorithm is a sequence of computational steps that transform the input into the output.



(ex) The sorting problem
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
Output: A permutation (reordering) $\langle a_1', a_2', \ldots, a_n' \rangle$ of the input sequence such that $a_1' \le a_2' \le \ldots \le a_n'$

- An algorithm is said to be correct if, for every input instance, it halts with the correct output.

- An algorithm can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a precise description of the computational procedure to be followed.

- Algorithms devised to solve the same problem often different in their efficiency. The most straight forward and simple algorithms for solving a given problem is often not the most efficient.

- Designing more efficient algorithms frequently requires utilizing more sophisticated data structures and familiarity with certain concepts from mathematics such as functions, sets, various summation formulas, and so forth.

-Why do you need to study algorithms?

(1) You have to know a standard set of important algorithms from different areas of computing.
⇒ You should be able to design new algorithms and analyze their efficiency.

(2) The study of algorithms has come to be recognized as the cornerstone of computer science. ⇒ Computer programs would not exist without algorithms.

(3) Specific algorithm design techniques can be interpreted as problem-solving strategies that can be useful regardless of whether a computer is involved.

(Note) 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.

The Present

- It is occasionally necessary to relax the requirements of an algorithm.
- Many algorithms currently in use are not general, deterministic, or even finite.

(ex) - An operating system is better thought of as a program that never terminate.

- Algorithms written for more than one processor are rarely deterministic.
- Many practical problems are too difficult to be solved efficiently, and compromises either in generality or correctness are necessary.

The Future

- There can be good reasons to extend the tradinal notion of an algorithm and computation, and, indeed, many areas of computer science require such extended computational models. Computer games, cryptography, and computational geometry are unthinkable without randomness, and distributed computation is not deterministic.
- Quantum Computing

  : In a quantum computer, the fundamental unit of information is a quantum bit or qubit. Like a bit, a qubit has values 0 and 1, but it can also exist in an intermediate state.
- DNA Computing

  : DNA encodes genetic information as a string of chemicals, called bases, typically denoted A, C, G, and T. In DNA computing bits (0 and 1) are replaced by the bases A, C, G, and T.

Chapter 2. Getting Started

(Note) In this book, we shall typically describe algorithms as programs written in a pseudocode that is similar in many respects to C, Pascal, or Java.
⇒ Read pseudocode conventions (p19-20)

A pseudocode is a mixture of a natural language and programming language-like constructs.

- Algorithm design and analysis process

(1) Understanding the problem
(2) Ascertaining the capabilities of a computational device
(Sequential or parallel algorithms)
(3) Choosing between exact and approximate problem solving
(Approximation due to the nature of problems or speed up purpose)
(4) Deciding on appropriate data structures
(5) Algorithm design techniques
(6) Methods of specifying an algorithm (Pseudocode, flowchart, and etc.)
(7) Proving an algorithm's correctness
(8) Analyzing an algorithm (Time efficiency, space efficiency, and etc.)
(9) Coding an algorithm

2.2. Analyzing algorithms

- Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. (ex. Memory, communication bandwidth, and etc.)

- Most often it is computational time that we want to measure.
(⇐ Resources are usually enough due to the improvement of computers)

- The running time of an algorithm on a particular input is the number of primitive operations executed.

- A constant amount of time is required to execute each line of our pseudocode.
(i.e. Assume that each execution of the i-th line take time c_i, c_i is a constant)

(ex) Insertion_Sort (A)

```
                                                          cost    times
1 for j←2 to length[A]                                    $c_1$     n = $(n-2+1)+1$
2    do key←A[j]                                          $c_2$     n-1
3       //Insert A[j] into the sorted sequence A[1..j-1]
4       i←j-1                                             $c_4$     n-1
5       while i>0 and A[i]>key                            $c_5$     $\Sigma_{j=2}^{n} t_j$
6            do A[i+1]←A[i]                               $c_6$     $\Sigma_{j=2}^{n} (t_j-1)$
7                i←i-1                                    $c_7$     $\Sigma_{j=2}^{n} (t_j-1)$
8       A[i+1]←key                                        $c_8$     n-1
```

※ t_j is the number of times the while loop test in line 5 is executed for that value of j

cost: time of each statement
times: number of times each statement is executed.

⇒ The running time of Insertion_Sort
$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j-1) + c_7\sum_{j=2}^{n}(t_j-1) + c_8(n-1)$

The best case occurs if the array is already sorted
(i.e. $A[i] \leq$ key for $j = 2, 3, \ldots, n$ $\therefore$ $t_j = 1$ for $j = 2, 3, \ldots, n$)
$\Rightarrow T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$
$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$
$= an + b$ (a, b: constants)

The worst case occurs if the array is in reverse sorted order
(i.e. $t_j = j$ for $j = 2, 3, \ldots, n$ $\therefore$ $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ $\quad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$)

$\Rightarrow T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + c_6(\frac{n(n-1)}{2}) + c_7(\frac{n(n-1)}{2})$
$+ c_8(n-1)$
$= an^2 + bn + c$ (a, b, c: constants)

(Note) We shall usually concentrate on finding only the worst-case running time.

- Order of growth (or rate of growth)
  : Consider only the leading term of running time, since the low-order terms are relatively insignificant for large n. We also ignore the leading term's constant coefficient.

(ex) Insertion sort has a worst-case running time of $\Theta(n^2)$.
Insertion sort has a best-case running time of $\Theta(n)$.

- We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth.

2.3. Designing Algorithms

- There are many ways to design algorithms.

(1) Brute Force (ex. bubble sort, linear search, ...)
(2) Divide and Conquer (ex. merge sort, quick sort, ...)
(3) Decrease and Conquer (or Incremental Approach)
(ex. insertion sort, DFS, BFS, topological sorting, ...)
(4) Transform and Conquer (ex. Gaussian elimination, heap sort, ...)
(5) Dynamic Programming
(ex. matrix-chain multiplication, longest common subsequence, ...)
(6) Greedy Algorithms (ex. activity-section, Huffman codes, ...)

(ex) Merge sort using divide and conquer technique

The divide-and-conquer approach involves 3 steps:

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.

2. The smaller instances are solved (typically recursively, though sometimes a different algorithm is employed when instances become small enough).

3. If necessary, the solution obtained for the smaller instances are combined to get a solution to the original problem.

```
Merge_Sort (A, p, r)

1   if p<r                                 C1 × 1 = C1  = θ(1)
2     then q ← ⌊(p+r)/2⌋                   C2 × 1 = C2  = θ(1)
3          Merge_Sort (A, p, q)            T(n/2) × 1   = T(n/2)
4          Merge_Sort (A, q+1, r)          T(n/2) × 1   = T(n/2)
5          Merge (A, p, q, r)                             θ(n)
```

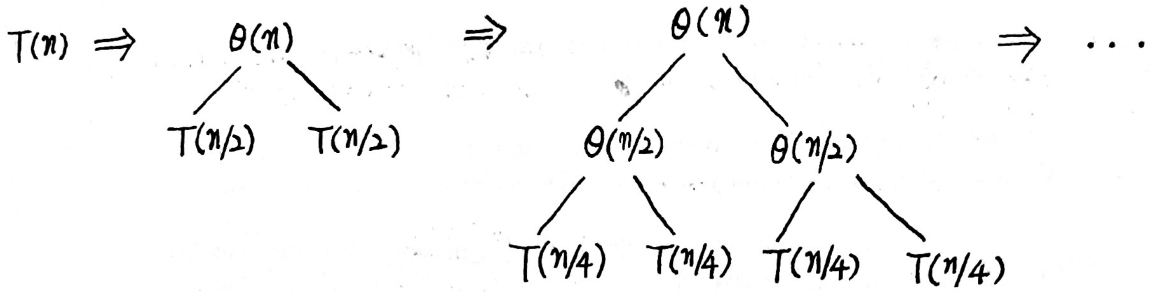
$$\Rightarrow T(n) = \begin{cases} \theta(1) & , \text{if } n= \\ 2T(n/2) + \theta(n) & , \text{if } n \end{cases}$$

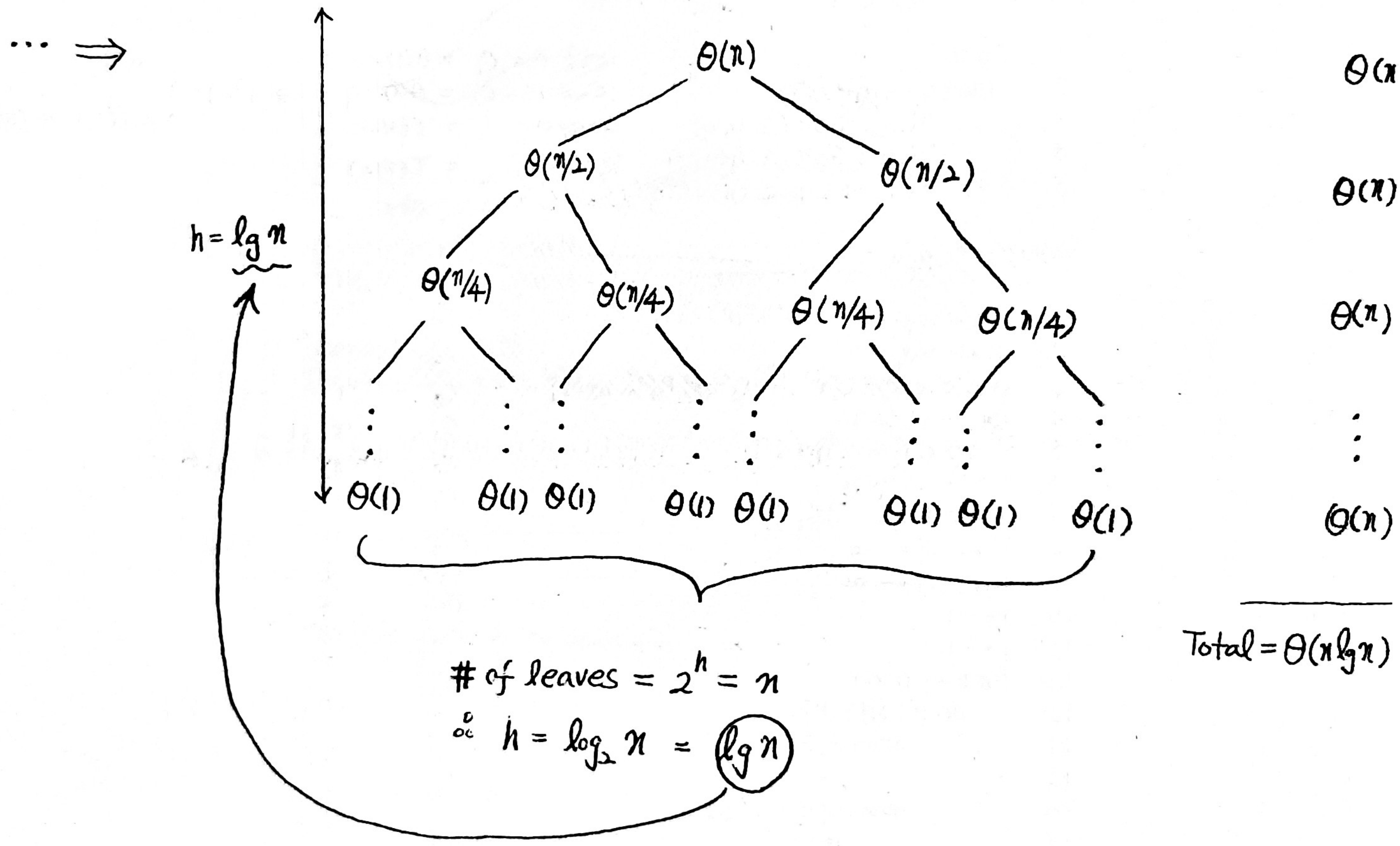
```
Merge (A, p, q, r)

1   n1 ← q-p+1                                  C1     1
2   n2 ← r-q                                    C2     1
3   create arrays L[1.. n1+1] and R[1.. n2+1]   C3     1
4   for i ← 1 to n1                             C4     n1+1
5       do L[i] ← A[p+i-1]                      C5     n1
6   for j ← 1 to n2                             C6     n2+1
7       do R[j] ← A[q+j]                        C7     n2
8   L[n1+1] ← ∞                                 C8     1
9   R[n2+1] ← ∞                                 C9     1
10  i ← 1                                       C10    1
11  j ← 1                                       C11    1
12  for k ← p to r                              C12    (r-p+1)+1 = n+1
13      do if L[i] ≤ R[j]                       C13    r-p+1 = n
14            then A[k] ← L[i]                  C14    ≤ n
15                 i ← i+1                      C15    ≤ n
16            else A[k] ← R[j]                  C16    ≤ n
17                 j ← j+1                      C17    ≤ n
```

$\Rightarrow T(n) \underset{\uparrow}{=} \theta(n)$ (∵) $\left.\begin{array}{l} 4 \sim 7 \Rightarrow \theta(n) \\ 12 \sim 17 \Rightarrow \theta(n) \end{array}\right\} \Rightarrow \theta(n)$

represents the ruming time
by using the order of growth

$$T(n) = \begin{cases} \Theta(1) & \text{, if } n=1 \\ 2T(n/2) + \Theta(n) & \text{, if } n>1 \end{cases}$$




. . . ⇒
h = lg n
Θ(n)
Θ(n/2) Θ(n/2)
Θ(n/4) Θ(n/4) Θ(n/4) Θ(n/4)
Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1) Θ(1)
Θ(n
Θ(n)
Θ(n)
Θ(n)

Total = $\Theta(n \lg n)$

# of leaves = $2^h = n$

$\therefore\ h = \log_2 n = \lg n$

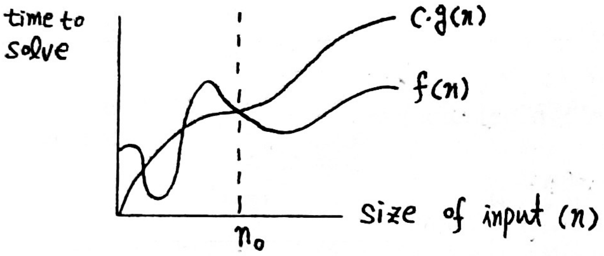
Chapter 3. Growth of Functions

3.1 Asymptotic Notation

The efficiency analysis framework concentrates on the order of growth. To compare and rank such orders of growth, computer scientists use three notations: O (big oh), Ω(big omega), and Θ(big theta).

(1) O (big oh)-notation: asymptotic upper bound

For a given function g(n),
$O(g(n)) = \{\ f(n) \mid \exists$ positive constants c and $\overset{\text{nonnegative integer}}{n_0}$ s.t. $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ }
Where, n represents size of input



(ex) Let $g(n) = n^2$ and $f(n) = 100n + 5$
Then $100n + 5 = O(n^2)$ (Note: O(g(n)) represents a set of functions
$\Rightarrow f(n) \in O(g(n)) \equiv f(n) = O(g(n))$)
$(\because)$ Take $n_0 = 1$, $c = 105$
Then $0 \leq 100n + 5 \leq 100n + 5n = 105n$ (for all $n \geq 1$) $\leq 105 n^2$

(Note) The definition gives us a lot of freedom in choosing specific values for c and n_0.

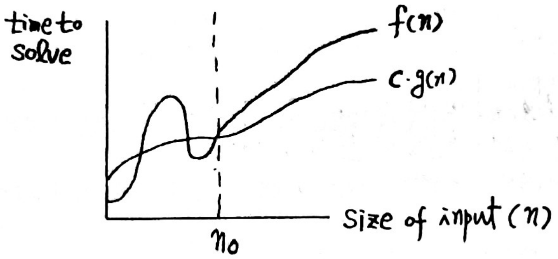
(ex) Let $g(n) = n^3$ and $f(n) = 2n^2$
Then $2n^2 = O(n^3)$
$(\because)$ Take $n_0 = 2$, $c = 1$
Then $0 \leq 2n^2 \leq 1n^3$ (for all $n \geq 2$)

(ex) Let $g(n) = n^3/2$ and $f(n) = 2n^2$
Then $2n^2 = O(n^3/2)$
$(\because)$ Take $n_0 = 2$, $c = 2$
Then $0 \leq 2n^2 \leq 2(n^3/2)$ (for all $n \geq 2$)

(ex) Let $g(n) = n^2+4$ and $f(n) = n^2+3$
Then $n^2+3 = O(n^2+4)$
$(\because)$ Take $n_0 = 1$, $c = 1$
Then $0 \leq n^2+3 \leq 1(n^2+4)$ (for all $n \geq 1$)

(2) Ω(big omega)-notation: asymptotic lower bound

For a given function g(n),
$\Omega(g(n)) = \{\ f(n) \mid \exists$ positive constants c and $\overset{\text{nonnegative integer}}{n_0}$ s.t. $0 \le cg(n) \le f(n)$ for all $n \ge n_0\ \}$
Where, n represents size of input



(ex) $n^3 = \Omega(n^2)$ (Note: Ω(g(n)) represents a set of functions
$\Rightarrow f(n) \in \Omega(g(n)) \equiv f(n) = \Omega(g(n))$)
($\because$) $c = 1, n_0 = 0 \Rightarrow 0 \le 1n^2 \le n^3$ for all $n \ge 0$

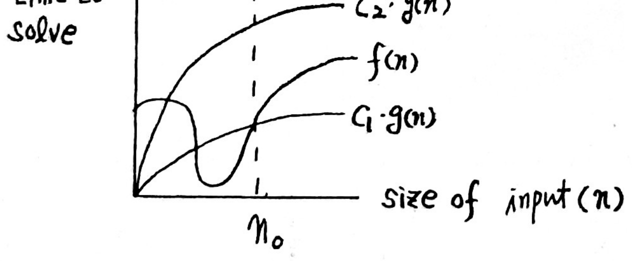
(ex) $n^3 = \Omega(2n^2)$
($\because$) $c = 1, n_0 = 2 \Rightarrow 0 \le (1)2n^2 \le n^3$ for all $n \ge 2$

(ex) $n^2 = \Omega(2n^2)$
($\because$) $c = ½, n_0 = 0 \Rightarrow 0 \le (½)2n^2 \le n^2$ for all $n \ge 0$

(ex) $\sqrt{n} = \Omega(\log\ n)$
($\because$) $c = 1, n_0 = 16 \Rightarrow 0 \le (1)\log\ n \le \sqrt{n}$ for all $n \ge 16$

(3) Θ(big theta)-notation: asymptotic tight bound

For a given function g(n),
$\Theta(g(n)) = \{\ f(n) \mid \exists$ positive constants c_1, c_2, and $\overset{\text{nonnegative integer}}{n_0}$ s.t. $0 \le c_1g(n) \le f(n) \le c_2g(n)$
for all $n \ge n_0\ \}$
Where, n represents size of input



(ex) $(\frac{1}{2})n(n-1) = \Theta(n^2)$ (Note: $\Theta(g(n))$ represents a set of functions $\Rightarrow f(n) \in \Theta(g(n)) \equiv f(n) = \Theta(g(n))$)

($\because$) (the right inequality)

$(\frac{1}{2})n(n-1) = (\frac{1}{2})n^2 - (\frac{1}{2})n \le (\frac{1}{2})n^2$ for all $n \ge 0$

(the left inequality)

$(\frac{1}{2})n(n-1) = (\frac{1}{2})n^2 - (\frac{1}{2})n \ge (\frac{1}{2})n^2 - (\frac{1}{2})n(\frac{1}{2})n$ (for all $n \ge 2$) $= (\frac{1}{4})n^2$

Hence, take $c_1 = \frac{1}{4}$, $c_2 = \frac{1}{2}$ and $n_0 = 2$

Then, $0 \le (\frac{1}{4})n^2 \le (\frac{1}{2})n(n-1) \le (\frac{1}{2})n^2$ for all $n \ge 2$

(ex) $(\frac{1}{2})n^2 - 3n = \Theta(n^2)$ (Note: $\Theta(g(n))$ represents a set of functions $\Rightarrow f(n) \in \Theta(g(n)) \equiv f(n) = \Theta(g(n))$)

($\because$) $0 \le c_1 n^2 \le (\frac{1}{2})n^2 - 3n \le c_2 n^2$

$0 \le c_1 \le (\frac{1}{2}) - 3/n \le c_2$

cannot be 0 ? $\frac{1}{2} - \frac{3}{7} = \frac{7-6}{14} = \frac{1}{14}$ ← Take $c_1 = \frac{1}{14}$ $c_2 = \frac{1}{2}$ and $n_0 = 7$

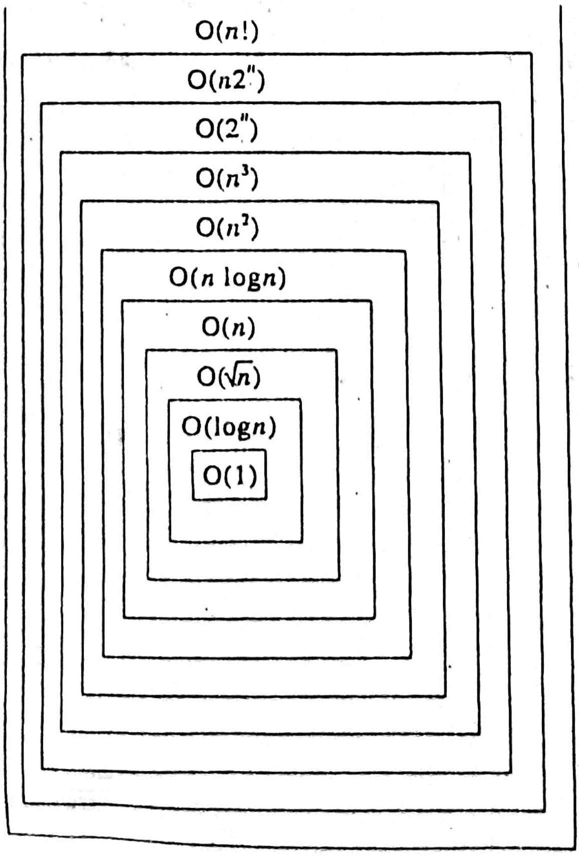
Then, $0 \le \frac{1}{14}n^2 \le (\frac{1}{2})n^2 - 3n \le (\frac{1}{2})n^2$ for all $n \ge 7$

(Note) We can always determine the relative growth rates of two functions f(n) and g(n) by comparing $\lim_{n\to\infty} \frac{f(n)}{g(n)}$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \Rightarrow f(n) = o(g(n)) \\ c\ (\neq 0) & \Rightarrow f(n) = \Theta(g(n)) \\ \infty & \Rightarrow g(n) = o(f(n)) \\ \text{oscillates} & \Rightarrow \text{no relation} \end{cases}$$

Where, $o(p(n))$ means $O(p(n))$ but not $\Theta(p(n))$.

※ Basic Asymptotic Efficiency Classes (: increasing order)



⊙ Examples of Algorithm Analysis

termines the
lue of the
rgest element
a given array

①
```
Algorithm MaxElement (A[0..n-1])
    maxval ← A[0]                      C1   1
    for i ← 1 to n-1 do                C2   [(n-1)-1+1]+1 = n
        if A[i] > maxval               C3   n-1
            maxval ← A[i]              C4   ≤ n-1
    return maxval                      C5   1
```

$\therefore T(n) = \theta(n)$

ecks whether all
e elements in
given array
e distinct.

②
```
Algorithm UniqueElements (A[0..n-1])
    for i ← 0 to n-2 do                        C1   [(n-2)-0+1]+1 = n
        for j ← i+1 to n-1 do                  C2   $\sum_{i=0}^{n-2}[(n-1)-(i+1)+1+1]$
            if A[i] = A[j] return false        C3   $\sum_{i=0}^{n-2}[(n-1)-(i+1)+1]$
    return true                                C4   1
```

$\therefore T(n) = \theta(n^2)$

$$(\because) \sum_{i=0}^{n-2}[(n-1)-(i+1)+1+1]$$
$$= \sum_{i=0}^{n-2}(n-i) = \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i = n\cdot(n-1) + \frac{(n-2)(n-1)}{2} = \theta(n^2)$$

ltiplies two
uare matrices
order n by
definition-based
gorithm.

③
```
Algorithm MatrixMultiplication (A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])
    for i ← 0 to n-1 do                          C1   [(n-1)-0+1]+1 = n+1
        for j ← 0 to n-1 do                      C2   $\sum_{i=0}^{n-1}[(n-1)-0+1+1]$
            C[i,j] ← 0.0                         C3   $\sum_{i=0}^{n-1}[(n-1)-0+1]$
            for k ← 0 to n-1 do                  C4   $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}[(n-1)-0+1+1]$
                C[i,j] ← C[i,j] + A[i,k]*B[k,j]  C5   $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}[(n-1)-0+1]$
    return C                                     C6   1
```

$\therefore T(n) = \theta(n^3)$

$$(\because) \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}[(n-1)-0+1+1]$$
$$= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(n+1) = \sum_{i=0}^{n-1}[n\cdot n + n] = \sum_{i=0}^{n-1}(n^2+n) = n^3+n^2 = \theta(n^3)$$