VII. Algorithm Complexity (Chap. 7)

Measuring the Efficiency of Algorithms: (§ 7.4)

1. What to measure?
   Space utilization: amount of memory required
   Time efficiency: amount of time required to process the data.

   — Depends on many factors: size of input, speed of machine, quality of source code, quality of compiler.

   Since most of these factors vary from one machine/compiler to another, we count the number of times instructions are executed. Thus, we measure computing time as:

   T(n) = the computing time of an algorithm for input of size n
   = the number of times the instructions are executed.

2. Example: See ALGORITHM TO CALCULATE MEAN on page 350

```
/*  Algorithm to find the mean of n real numbers.
    Receive:  An integer n ≥ 1 and an array x[0], . . . , x[n–1] of real numbers
    Return:   The mean of x[0], . . . , x[n–1]
-----------------------------------------------------------------------------------*/
1.  Initialize sum to 0.
2.  Initialize index variable i to 0.
3.  While i < n do the following:
4.      a.  Add x[i] to sum.
5.      b.  Increment i by 1.
6.  Calculate and return mean = sum / n .
```

   $T(n) = 3n + 4$

3. Definition of "big-O notation: The computing time of an algorithm is said to have ***order of magnitude f(n)***, written **T(n) is O(f(n))**
   if there is some constant C such that
   $T(n) \le C \cdot f(n)$ for all sufficiently large values of n.

   We also say, the ***complexity*** of the algorithm is O(f(n)).

   Example: For the Mean-Calculation Algorithm:
   T(n) is O(n)

4. The arrangement of the input items may affect the computing time. For example, it may take more time to sort a list of element that are nearly in order than for one that are completely out of order. We might measure it in the *best case* or in the *worst case* or try for the *average*. Usually best-case isn't very informative, average-case is too difficult to calculate; so we usually measure worst-case performance.

5. Example:

a. LINEAR SEARCH ALGORITHM on p. 354

```
/* Algorithm to perform a linear search of the list a[0], . . . , a[n – 1].
   Receive: An integer n and a list of n elements stored in array elements
            a[0], . . . , a[n – 1], and item of the same  type as the array elements.
   Return:  found = true and loc = position of item if the search is successful;
            otherwise, found is false.
-------------------------------------------------------------------------------------*/
1. Set found = false.
2. Set loc = 0.
3. While loc < n and not found do the following:
4.      If item = a[loc] then        // item found
5.           Set found = true.
6.      Else                         // keep searching *)
             Increment loc by 1.
```

Worst case: Item not in the list:
$T_L(n)$ is $O(n)$

b. BINARY SEARCH ALGORITHM on p. 355

```
/* Algorithm to perform a binary search of the list a[0], . . . , a[n – 1]
   in which the items are in ascending order.
   Receive: An integer n and a list of n elements in ascending order
            stored in array elements a[0], . . . , a[n – 1], and item of
            the same type as the array elements.
   Return:  found = true and loc = position of item if the search is
            successful; otherwise, found is false.
--------------------------------------------------------------------------------*/
1. Set found = false.
2. Set first = 0.
3. Set last = n – 1.
4. While first < last and not found do the following:
5.      Calculate loc = (first + last) / 2.
6.      If item < a[loc] then
7.           Set last = loc – 1.       // search first half
8.      Else if item > a[loc] then
9.           Set first = loc + 1.          // search last half
10.     Else
             Set found = true.   // item found
```

Worst case: Item not in the list:
$T_B(n) = O(\log_2 n)$

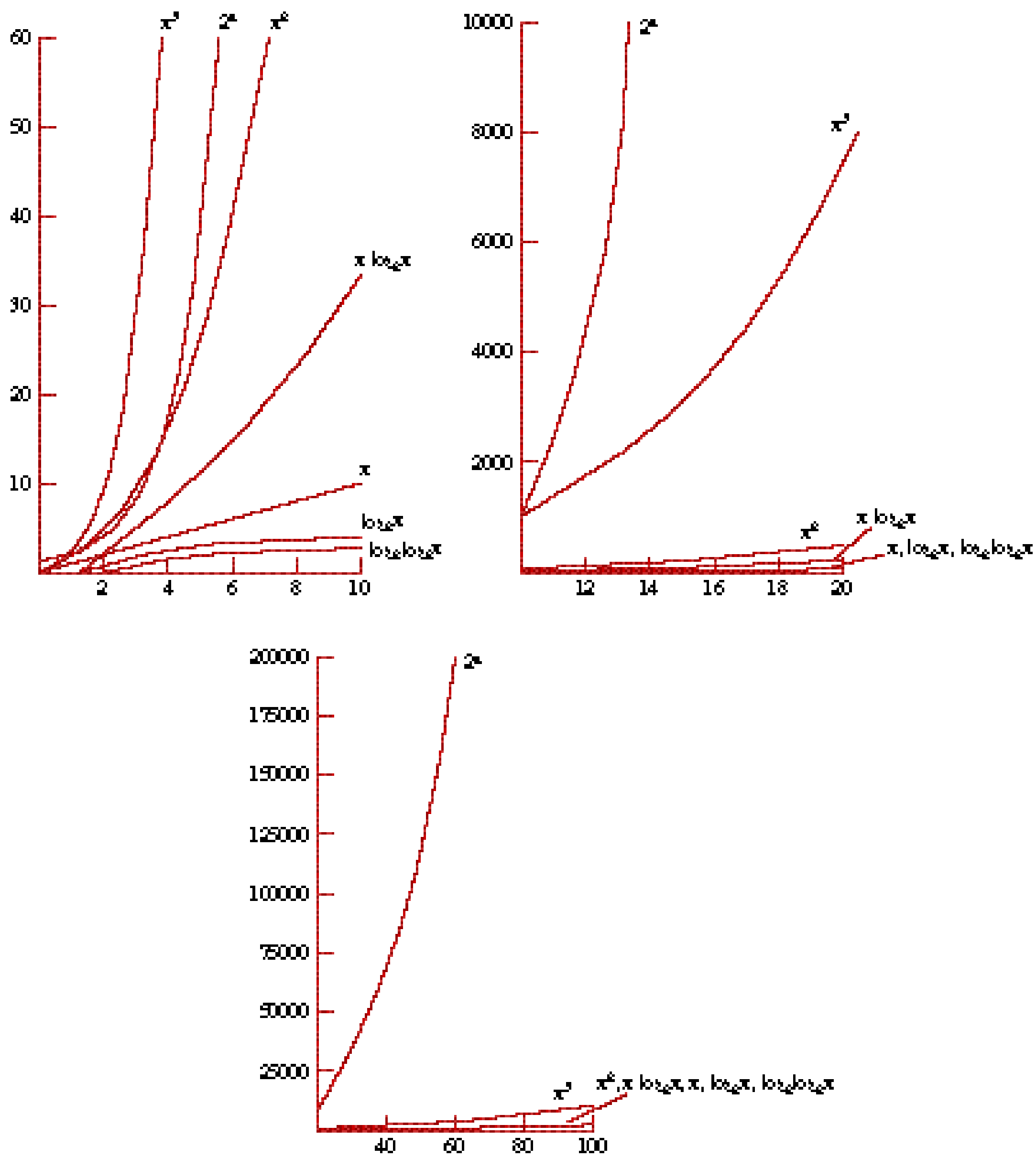
6. Commonly-used computing times:

$O(\log_2\log_2 n)$, $O(\log_2 n)$, $O(n)$, $O(n\log_2 n)$, $O(n^2)$, $O(n^3)$, and $O(2^n)$

See the table on p. 7-43 and graphs on p. 7-44 for a comparison of these.

Table 7.1 Common Computing Time Functions

$\log_2\log_2 n$	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
—	0	1	0	1	1	2
0	1	2	2	4	8	4
1	2	4	8	16	64	16
1.58	3	8	24	64	512	256
2	4	16	64	256	4096	65536
2.32	5	32	160	1024	32768	4294967296
2.6	6	64	384	4096	2.6×10^5	1.85×10^{19}
3	8	256	2.05×10^3	6.55×10^4	1.68×10^7	1.16×10^{77}
3.32	10	1024	1.02×10^4	1.05×10^6	1.07×10^9	1.8×10^{308}
4.32	20	1048576	2.1×10^7	1.1×10^{12}	1.15×10^{18}	6.7×10^{315652}



7. Computing times of Recursive Algorithms

Have to solve a recurrence relation.

Example: Towers of Hanoi

```
void Move(int n,
          char source, char destination, char spare)
{
  if (n <= 1)                      // anchor
    cout << "Move the top disk from " << source
         << " to " << destination << endl;
  else
  {                                // inductive case
    Move(n-1, source, spare, destination);
    Move(1, source, destination, spare);
    Move(n-1, spare, destination, source);
  }
}
```

$T(n) = O(2^n)$