제1장 Basic Concepts

경북대학교 임경식 교수

1.1 System life cycle

Software development process

- Requirement specification
- Problem analysis
- System design: data structures & relevant algorithms
- Refinement & Coding
- Verification

1.2 Dynamic memory allocation

```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc(sizeof(float));
*pi = 1024;
*pf = 3.14;
printf("an integer = %d, a float = %f\n", *pi, *pf);
free(pi);
free(pf);
```

Program 1.1: Allocation and deallocation of memory

```
#define MALLOC(p,s) \
   if (!((p) = malloc(s))) {\
      fprintf(stderr, "Insufficient memory"); \
      exit(EXIT_FAILURE);\
   }

MALLOC(pi, sizeof(int));
MALLOC(pf, sizeof(float));
```

1.3 Algorithm specification

Definition: An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

(1) **Input.** There are zero or more quantities that are externally supplied.

(2) **Output.** At least one quantity is produced.

(3) **Definiteness.** Each instruction is clear and unambiguous.

(4) **Finiteness.** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

(5) **Effectiveness.** Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible. □

Selection sort

Example 1.1 [***Selection sort***]**:** Suppose we must devise a program that sorts a set of $n \geq 1$ integers. A simple solution is given by the following:

From those integers that are currently unsorted, find the smallest and place it next in the sorted list.

- Although this statement adequately describes the sorting problem, it is not an algorithm since it leaves several unanswered questions.
- For example, it does not tell us where and how the integers are initially stored, or where we should place the result.

Selection sort

```
for (i = 0; i < n; i++) {
  Examine list[i] to list[n-1] and suppose that the
  smallest integer is  at list[min];

  Interchange list[i] and list[min];
}
```

Program 1.2: Selection sort algorithm

Selection sort

```
void swap(int *x, int *y)
{/* both parameters are pointers to ints */
   int temp = *x;    /* declares temp as an int and assigns
                        to it the contents of what x points to */
   *x = *y; /* stores what y points to into the location
               where x points */
   *y = temp; /* places the contents of temp in location
                 pointed to by y */
}
```

Program 1.3: Swap function

```
swap(&a, &b);
```

```
#define SWAP(x,y,t) ((t) = (x), (x) = (y), (y) = (t))
```

Selection sort

```
#include <stdio.h>
#include <math.h>
#define MAX_SIZE 101
#define SWAP(x,y,t) ((t) = (x), (x)= (y), (y) = (t))
void sort(int [],int); /*selection sort */
void main(void)
{
   int i,n;
   int list[MAX_SIZE];
   printf("Enter the number of numbers to generate: ");
   scanf("%d",&n);
   if( n < 1 || n > MAX_SIZE) {
     fprintf(stderr, "Improper value of n\n");
     exit(EXIT_FAILURE);
   }
   for (i = 0; i < n; i++) {/*randomly generate numbers*/
      list[i] = rand() % 1000;
      printf("%d  ",list[i]);
   }
   sort(list,n);
   printf("\n Sorted array:\n ");
   for (i = 0; i < n; i++) /* print out sorted numbers */
      printf("%d  ",list[i]);
   printf("\n");
}
```

Selection sort

```
void sort(int list[],int n)
{
   int i, j, min, temp;
   for (i = 0; i < n-1; i++)  {
      min = i;
      for (j = i+1; j < n; j++)
         if (list[j] < list[min])
            min = j;
      SWAP(list[i],list[min],temp);
   }
}
```

Binary search

Example 1.2 [*Binary search*]: Assume that we have $n \geq 1$ distinct integers that are already sorted and stored in the array *list*. That is, *list* [0] $\leq$ *list* [1] $\leq \cdots \leq$ *list* [$n-1$]. We must figure out if an integer *searchnum* is in this list. If it is we should return an index, i, such that *list*[i] = *searchnum*. If *searchnum* is not present, we should return -1. Since the list is sorted we may use the following method to search for the value.

Let *left* and *right*, respectively, denote the left and right ends of the list to be searched. Initially, *left* = 0 and *right* = $n-1$. Let *middle* = (*left*+*right*)/2 be the middle position in the list. If we compare *list* [*middle*] with *searchnum*, we obtain one of three results:

(1) **searchnum < list[middle].** In this case, if *searchnum* is present, it must be in the positions between 0 and *middle* − 1. Therefore, we set *right* to *middle* − 1.

(2) **searchnum = list[middle].** In this case, we return *middle*.

(3) **searchnum > list[middle].** In this case, if *searchnum* is present, it must be in the positions between *middle* + 1 and n − 1. So, we set *left* to *middle* + 1.

Binary search

```
#define COMPARE(x,y) (((x) < (y)) ? -1: ((x) == (y))? 0: 1)
```

```
int binsearch(int list[], int searchnum, int left,
                                         int right)
{/* search list[0] <= list[1] <=  · · ·  <= list[n-1] for
 searchnum. Return its position if found. Otherwise
 return -1 */
   int  middle;
   while (left <= right)  {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: left = middle + 1;
                  break;
         case 0 : return middle;
         case 1 : right = middle - 1;
      }
   }
   return -1;
}
```

Program 1.7: Searching an ordered list

Binary search

```
int binsearch(int list[], int searchnum, int left,
                                         int right)
{/* search list[0] <= list[1] <= ... <= list[n-1] for
    searchnum. Return its position if found. Otherwise
    return -1 */
   int middle;
   if (left <= right) {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case -1: return
            binsearch(list, searchnum, middle + 1, right);
         case 0 : return middle;
         case 1 : return
            binsearch(list, searchnum, left, middle - 1);
      }
   }
   return -1;
}
```

Program 1.8: Recursive implementation of binary search

Binary search

Frequently computer science students regard recursion as a mystical technique that is useful for only a few special problems such as computing factorials or Ackermann's function. This is unfortunate because any function that we can write using assignment, **if-else**, and **while** statements can be written recursively. Often this recursive function is easier to understand than its iterative counterpart.

1.4 Data abstraction

Definition: A *data type* is a collection of *objects* and a set of *operations* that act on those objects. □

Whether your program is dealing with predefined data types or user-defined data types, these two aspects must be considered: objects and operations. For example, the data type **int** consists of the objects {0, +1, −1, +2, −2, ··· , INT_MAX, INT_MIN}, where INT_MAX and INT_MIN are the largest and smallest integers that can be represented on your machine. (They are defined in *limits.h*.) The operations on integers are many, and would certainly include the arithmetic operators +, −, *, /, and %. There is also testing for equality/inequality and the operation that assigns an integer to a variable. In all of these cases, there is the name of the operation, which may be a prefix operator, such as *atoi*, or an infix operator, such as +. Whether an operation is defined in the language or in a library, its name, possible arguments and results must be specified.

1.4 Data abstraction

Definition: An *abstract data type (ADT)* is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operations. □

(1) **Creator/constructor**: These functions create a new instance of the designated type.

(2) **Transformers**: These functions also create an instance of the designated type, generally by using one or more other instances. The difference between constructors and transformers will become more clear with some examples.

(3) **Observers/reporters**: These functions provide information about an instance of the type, but they do not change the instance.

1.4 Data abstraction

ADT *NaturalNumber* is

objects: an ordered subrange of the integers starting at zero and ending at the maximum integer (*INT_MAX*) on the computer

functions:

for all $x, y \in$ *NaturalNumber*; *TRUE*, *FALSE* $\in$ *Boolean*
and where +, −, <, and == are the usual integer operations

NaturalNumber Zero( )	::=	0
Boolean IsZero(x)	::=	**if** (x) **return** *FALSE*
		else return *TRUE*
Boolean Equal(x, y)	::=	**if** ($x == y$) **return** *TRUE*
		else return *FALSE*
NaturalNumber Successor(x)	::=	**if** ($x ==$ *INT_MAX*) **return** x
		else return $x + 1$
NaturalNumber Add(x, y)	::=	**if** (($x + y$) <= *INT_MAX*) **return** $x + y$
		else return *INT_MAX*
NaturalNumber Subtract(x, y)	::=	**if** ($x < y$) **return** 0
		else return $x - y$

end *NaturalNumber*

ADT 1.1: Abstract data type *NaturalNumber*

1.5 Performance analysis

Definition: The *space complexity* of a program is the amount of memory that it needs to run to completion. The *time complexity* of a program is the amount of computer time that it needs to run to completion. □

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE],
                  int c[][MAX_SIZE], int rows, int cols)
{
   int i, j;
   for (i = 0; i < rows; i++)
      for (j = 0; j < cols; j++)
         c[i][j] = a[i][j] + b[i][j];
}
```

Program 1.16: Matrix addition

1.5 Performance analysis

Definition: [Big "oh"] $f(n) = O(g(n))$ (read as "f of n is big oh of g of n") iff (if and only if) there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all n, $n \geq n_0$. □

Example 1.15: $3n + 2 = O(n)$ as $3n + 2 \leq 4n$ for all $n \geq 2$. $3n + 3 = O(n)$ as $3n + 3 \leq 4n$ for all $n \geq 3$. $100n + 6 = O(n)$ as $100n + 6 \leq 101n$ for $n \geq 10$. $10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for $n \geq 5$. $1000n^2 + 100n - 6 = O(n^2)$ as $1000n^2 + 100n - 6 \leq 1001n^2$ for $n \geq 100$. $6*2^n + n^2 = O(2^n)$ as $6*2^n + n^2 \leq 7*2^n$ for $n \geq 4$. $3n + 3 = O(n^2)$ as $3n + 3 \leq 3n^2$ for $n \geq 2$. $10n^2 + 4n + 2 = O(n^4)$ as $10n^2 + 4n + 2 \leq 10n^4$ for $n \geq 2$. $3n + 2 \neq O(1)$ as $3n + 2$ is not less than or equal to c for any constant c and all n, $n \geq n_0$. $10n^2 + 4n + 2 \neq O(n)$. □

We write $O(1)$ to mean a computing time which is a constant. $O(n)$ is called linear, $O(n^2)$ is called quadratic, $O(n^3)$ is called cubic, and $O(2^n)$ is called exponential. If an algorithm takes time $O(\log n)$ it is faster, for sufficiently large n, than if it had taken $O(n)$. Similarly, $O(n \log n)$ is better than $O(n^2)$ but not as good as $O(n)$. These seven computing times, $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, and $O(2^n)$ are the ones we will see most often in this book.

1.5 Performance analysis

Definition: [Omega] $f(n) = \Omega(g(n))$ (read as "f of n is omega of g of n") iff there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n, n \geq n_0$. □

Example 1.16: $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for $n \geq 1$ (actually the inequality holds for $n \geq 0$ but the definition of Ω requires an $n_0 > 0$). $3n + 3 = \Omega(n)$ as $3n + 3 \geq 3n$ for $n \geq 1$. $100n + 6 = \Omega(n)$ as $100n + 6 \geq 100n$ for $n \geq 1$. $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for $n \geq 1$. $6*2^n + n^2 = \Omega(2^n)$ as $6*2^n + n^2 \geq 2^n$ for $n \geq 1$. Observe also that $3n + 3 = \Omega(1)$; $10n^2 + 4n + 2 = \Omega(n)$; $10n^2 + 4n + 2 = \Omega(1)$; $6*2^n + n^2 = \Omega(n^{100})$; $6*2^n + n^2 = \Omega(n^{50.2})$; $6*2^n + n^2 = \Omega(n^2)$; $6*2^n + n^2 = \Omega(n)$; and $6*2^n + n^2 = \Omega(1)$. □

As in the case of the "big oh" notation, there are several functions $g(n)$ for which $f(n) = \Omega(g(n))$. $g(n)$ is only a lower bound on $f(n)$. For the statement $f(n) = \Omega(g(n))$ to be informative, $g(n)$ should be as large a function of n as possible for which the statement $f(n) = \Omega(g(n))$ is true. So, while we shall say that $3n + 3 = \Omega(n)$ and that $6*2^n + n^2 = \Omega(2^n)$, we shall almost never say that $3n + 3 = \Omega(1)$ or that $6*2^n + n^2 = \Omega(1)$ even though both these statements are correct.

1.5 Performance analysis

Definition: [Theta] $f(n) = \Theta(g(n))$ (read as "f of n is theta of g of n") iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$. □

Example 1.17: $3n + 2 = \Theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$ and $3n + 2 \leq 4n$ for all $n \geq 2$, so c_1 3, $c_2 = 4$, and $n_0 = 2$. $3n + 3 = \Theta(n)$; $10n^2 + 4n + 2 = \Theta(n^2)$; $6*2^n + n^2 = \Theta(2^n)$; and $10*\log n + 4 = \Theta(\log n)$. $3n + 2 \neq \Theta(1)$; $3n + 3 \neq \Theta(n^2)$; $10n^2 + 4n + 2 \neq \Theta(n)$; $10n^2 + 4n + 2 \neq \Theta(1)$; $6*2^n + n^2 \neq \Theta(n^2)$; $6*2^n + n^2 \neq \Theta(n^{100})$; and $6*2^n + n^2 \neq \Theta(1)$. □

The theta notation is more precise than both the "big oh" and omega notations. $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper and lower bound on $f(n)$.

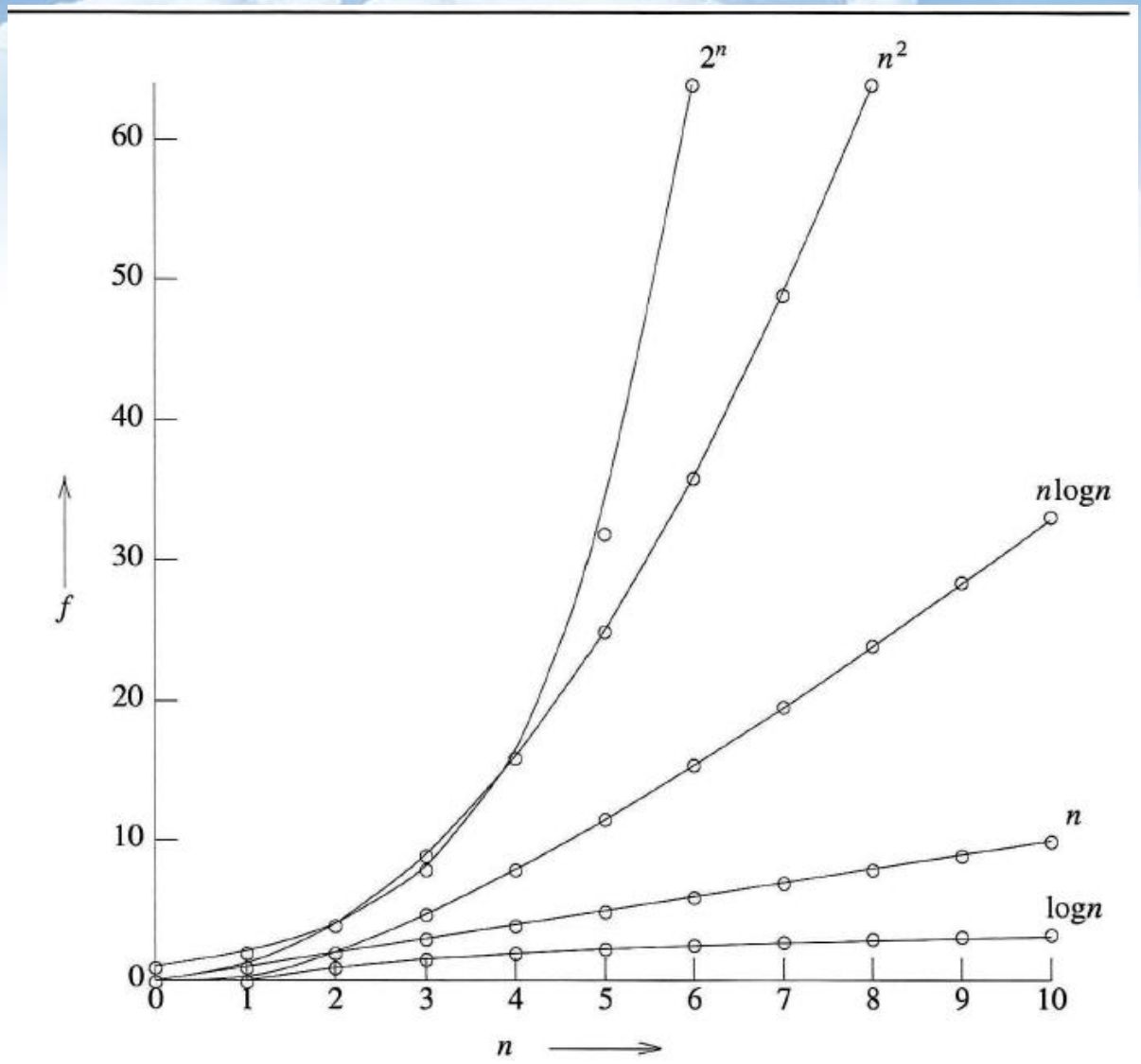
Notice that the coefficients in all of the $g(n)$'s used in the preceding three examples has been 1. This is in accordance with practice. We shall almost never find ourselves saying that $3n + 3 = O(3n)$, or that $10 = O(100)$, or that $10n^2 + 4n + 2 = \Omega(4n^2)$, or that $6*2^n + n^2 = \Omega(6*2^n)$, or that $6*2^n + n^2 = \Theta(4*2^n)$, even though each of these statements is true.

1.5 Performance analysis

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,768	4,294,967,296

Figure 1.7: Function values

1.5 Performance analysis



1.5 Performance analysis

	$f(n)$						
n	n	$n\log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 μs	.03 μs	.1 μs	1 μs	10 μs	10 s	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84 h	1 ms
30	.03 μ	.15 μ	.9 μ	27 μ	810 μ	6.83 d	1 s
40	.04 μs	.21 μs	1.6 μs	64 μs	2.56 ms	121 d	18 m
50	.05 μs	.28 μs	2.5 μs	125 μs	6.25 ms	3.1 y	13 d
100	.10 μs	.66 μs	10 μs	1 ms	100 ms	3171 y	$4*10^{13}$ y
10^3	1 μs	9.96 μs	1 ms	1 s	16.67 m	$3.17*10^{13}$ y	$32*10^{283}$ y
10^4	10 μs	130 μs	100 ms	16.67 m	115.7 d	$3.17*10^{23}$ y	
10^5	100 μs	1.66 ms	10 s	11.57 d	3171 y	$3.17*10^{33}$ y	
10^6	1 ms	19.92 ms	16.67 m	31.71 y	$3.17*10^7$ y	$3.17*10^{43}$ y	

μs = microsecond = 10^{-6} seconds; ms = milliseconds = 10^{-3} seconds
s = seconds; m = minutes; h = hours; d = days; y = years

Figure 1.9: Times on a 1-billion-steps-per-second computer

Thank You !

노력 없이 이룰 수 있는 것 아무것도 없다.