

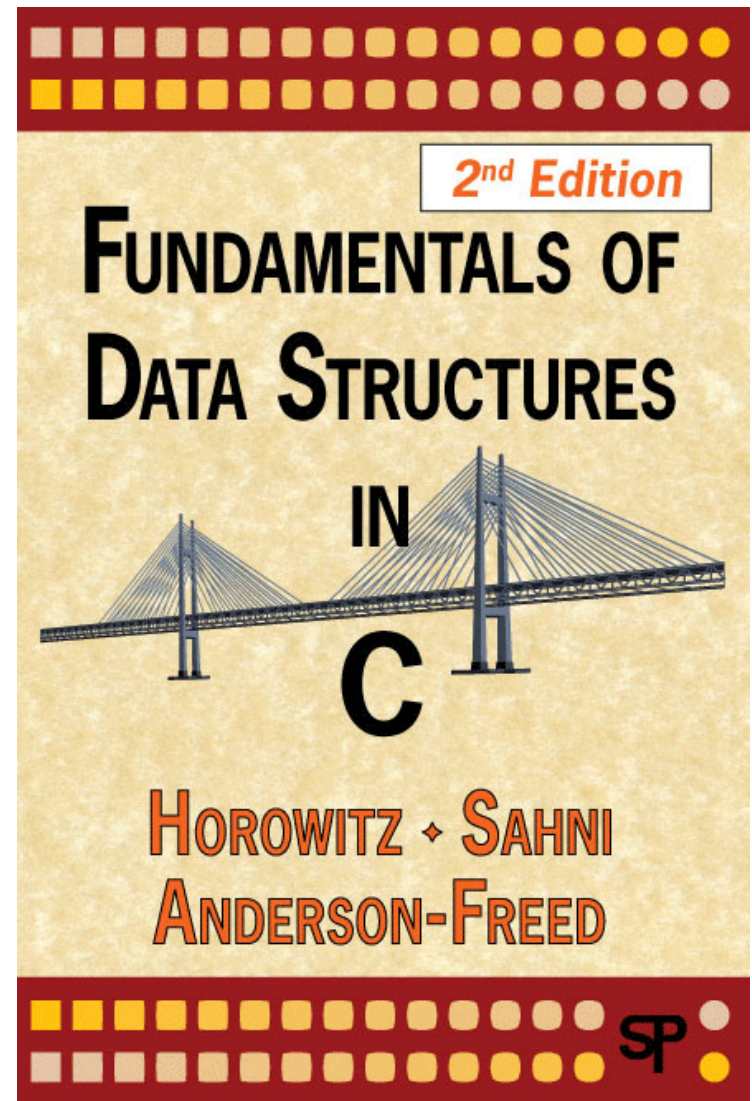


Data Structures

Books

Fundamentals of Data Structures in C, 2nd Edition.

(開發圖書，(02) 8242-3988)





Administration

Instructor:

- 曾學文 資工系副教授
- Office: Room 908
- Email: hwtseng@nchu.edu.tw
- Tel: 04-22840497 ext. 908
- <http://wccclab.cs.nchu.edu.tw/www/>

Office Hours:

- (Monday) 14:00~17:00; (Tuesday) 14:00~17:00.

Grade:

- Quiz 30%
- Homework 20%
- Midterm Exam 25%
- Final Exam 25%



Introductory

- Raise your hand is always welcome!
- No phone, walk, sleep, and late during the lecture time.
- Slides are not enough. To master the materials, page-by-page reading is necessary.



CHAPTER 1

BASIC CONCEPT

All the programs in this file are selected from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
“Fundamentals of Data Structures in C”,



How to create programs

- Requirements
- Analysis: **bottom-up vs. top-down**
- Design: **data objects and operations**
- Refinement and Coding
- Verification
 - Program Proving
 - Testing
 - Debugging



Algorithm

- Definition

An *algorithm* is a finite set of instructions that accomplishes a particular task.

- Criteria

- input
- output
- definiteness: clear and unambiguous
- finiteness: terminate after a finite number of steps
- effectiveness: instruction is basic enough to be carried out



Data Type

- Data Type

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

- Abstract Data Type (ADT)

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



Specification vs. Implementation

- Operation specification
 - function name
 - the types of arguments
 - the type of the results
- Implementation independent

*Structure 1.1: Abstract data type *Natural_Number*

structure *Natural_Number* 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 $x, y \in \text{Nat_Number}$; $TRUE, FALSE \in \text{Boolean}$
and where $+$, $-$, $<$, and $==$ are the usual integer operations.

Nat_Num Zero () ::= 0

Boolean Is_Zero(x) ::= if (x) return *FALSE*
else return *TRUE*

Nat_Num Add(x, y) ::= if ((x+y) <= *INT_MAX*) return x+y
else return *INT_MAX*

Boolean Equal(x,y) ::= if (x== y) return *TRUE*
else return *FALSE*

Nat_Num Successor(x) ::= if (x == *INT_MAX*) return x
else return x+1

Nat_Num Subtract(x,y) ::= if (x<y) return 0
else return x-y

end *Natural_Number*



Measurements

- Criteria
 - Is it correct?
 - Is it readable?
 - ...
- Performance Measurement (machine dependent)
- Performance Analysis (machine independent)
 - space complexity: storage requirement
 - time complexity: computing time

Space Complexity

$$S(P) = C + S_P(I)$$

- Fixed Space Requirements (C)

Independent of the characteristics of the inputs and outputs

- instruction space
- space for simple variables, fixed-size structured variable, constants

- Variable Space Requirements ($S_P(I)$)

depend on the instance characteristic I

- number, size, values of inputs and outputs associated with I
- recursive stack space, formal parameters, local variables, return address

***Program 1.10: Simple arithmetic function**

```
float abc(float a, float b, float c)
{
    return a + b + b * c + (a + b - c) / (a + b) + 4.00;
}
```

This function has only fixed space requirements $S_{abc}(I) = 0$

***Program 1.11: Iterative function for summing a list of numbers**

```
float sum(float list[ ], int n)
{
    float tempsum = 0;
    int i;
    for (i = 0; i < n; i++)
        tempsum += list [i];
    return tempsum;
}
```

$$S_{\text{sum}}(I) = 0$$

Recall: pass the address of the first element of the array & pass by value

***Program 1.12: Recursive function** for summing a list of numbers

```
float rsum(float list[ ], int n)
{
    if (n) return rsum(list, n-1) + list[n-1];
    return 0;
}
```

$$S_{\text{sum}}(I) = S_{\text{sum}}(n) = 12n$$

Assumptions:

***Figure 1.1:** Space needed for one recursive call of Program 1.12

Type	Name	Number of bytes
parameter: array pointer	list []	4
parameter: integer	n	4
return address:(used internally)		4 (unless a far address)
TOTAL per recursive call		12

Time Complexity

$$T(P) = C + T_P(I)$$

- Compile time (C)
independent of instance characteristics
- Run (execution) time T_P

- Definition $T_P(n) = c_a ADD(n) + c_s SUB(n) + c_l LDA(n) + c_{st} STA(n)$

A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

- Example

- $abc = a + b + b * c + (a + b - c) / (a + b) + 4.0$
- $abc = a + b + c$

Regard as the same unit
machine independent



Methods to compute the step count

- Introduce variable count into programs
- Tabular method
 - Determine the total number of steps contributed by each statement
step per execution × frequency
 - add up the contribution of all statements

Iterative summing of a list of numbers

*Program 1.13: Program 1.11 with count statements

```
float sum(float list[ ], int n)
{
    float tempsum = 0; count++; /* for assignment */
    int i;
    for (i = 0; i < n; i++) {
        count++;          /*for the for loop */
        tempsum += list[i]; count++; /* for assignment */
    }
    count++;          /* last execution of for */
    count++;          /* for return */
    return tempsum;
}
```

$2n + 3$ steps



*Program 1.14: Simplified version of Program 1.13

```
float sum(float list[ ], int n)
{
    float tempsum = 0;
    int i;
    for (i = 0; i < n; i++)
        count += 2;
    count += 3;
    return 0;
}
```

count final value is
 $2n + 3$

Recursive summing of a list of numbers

*Program 1.15: Program 1.12 with count statements added

```
float rsum(float list[ ], int n)
{
    count++;    /*for if conditional */
    if (n) {
        count++; /* for return and rsum invocation */
        return rsum(list, n-1) + list[n-1];
    }
    count++;
    return list[0];
}
```

$2n+2$

Matrix addition

*Program 1.16: Matrix addition

```
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 1.17: Matrix addition with count statements

```
void add(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE],
         int c[ ][MAX_SIZE], int row, int cols )
{
    int i, j;
    for (i = 0; i < rows; i++){
        count++; /* for i for loop */
        for (j = 0; j < cols; j++) {
            count++; /* for j for loop */
            c[i][j] = a[i][j] + b[i][j];
            count++; /* for assignment statement */
        }
        count++; /* last time of j for loop */
    }
    count++; /* last time of i for loop */
}
```

*Program 1.18: Simplification of Program 1.17

```
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++)
            count += 2;
            count += 2;
        }
    count++;
}
```

$2rows \times cols + 2rows + 1$

Suggestion: Interchange the loops when $rows \gg cols$

Tabular Method

*Figure 1.2: Step count table for Program 1.11

Iterative function to sum a list of numbers
steps/execution

Statement	s/e	Frequency	Total steps
float sum(float list[], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Recursive Function to sum of a list of numbers

***Figure 1.3:** Step count table for recursive summing function

Statement	s/e	Frequency	Total steps
float rsum(float list[], int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list, n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

Matrix Addition

*Figure 1.4: Step count table for matrix addition

Statement	s/e	Frequency	Total steps
Void add (int a[][MAX_SIZE] . . .)	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i = 0; i < row; i++)	1	rows+1	rows+1
for (j=0; j< cols; j++)	1	rows • (cols+1)	rows • cols+rows
c[i][j] = a[i][j] + b[i][j];	1	rows • cols	rows • cols
}	0	0	0
Total			2rows • cols+2rows+1

Exercise 1

*Program 1.19: Printing out a matrix

```
void print_matrix(int matrix[ ][MAX_SIZE], int rows, int cols)
{
    int i, j;
    for (i = 0; i < row; i++) {
        for (j = 0; j < cols; j++)
            printf("%d", matrix[i][j]);
        printf( "\n");
    }
}
```

$2n+2$

Exercise 2

*Program 1.20: Matrix multiplication function

```
void mult(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE], int c[ ][MAX_SIZE])
{
    int i, j, k;
    for (i = 0; i < MAX_SIZE; i++)
        for (j = 0; j < MAX_SIZE; j++) {
            c[i][j] = 0;
            for (k = 0; k < MAX_SIZE; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
}
```

$O(\text{MAX_SIZE})^3$

Exercise 3

*Program 1.21:Matrix product function

```
void prod(int a[ ][MAX_SIZE], int b[ ][MAX_SIZE], int c[ ][MAX_SIZE],
          int rowsa, int colsb, int colsa)
{
    int i, j, k;
    for (i = 0; i < rowsa; i++)
        for (j = 0; j < colsb; j++) {
            c[i][j] = 0;
            for (k = 0; k < colsa; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
}
```

???

Exercise 4

*Program 1.22:Matrix transposition function

```
void transpose(int a[ ][MAX_SIZE])
{
    int i, j, temp;
    for (i = 0; i < MAX_SIZE-1; i++)
        for (j = i+1; j < MAX_SIZE; j++)
            SWAP (a[i][j], a[j][i], temp);
}
```

???

Asymptotic Notation (O)

■ Definition

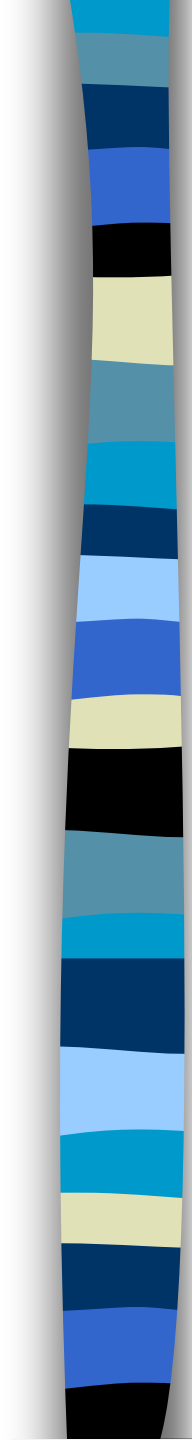
$f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all n , $n \geq n_0$.

■ Examples

- $3n+2=O(n)$ /* $3n+2 \leq 4n$ for $n \geq 2$ */
- $3n+3=O(n)$ /* $3n+3 \leq 4n$ for $n \geq 3$ */
- $100n+6=O(n)$ /* $100n+6 \leq 101n$ for $n \geq 10$ */
- $10n^2+4n+2=O(n^2)$ /* $10n^2+4n+2 \leq 11n^2$ for $n \geq 5$ */
- $6 \cdot 2^n + n^2 = O(2^n)$ /* $6 \cdot 2^n + n^2 \leq 7 \cdot 2^n$ for $n \geq 4$ */

Example

- Complexity of $c_1n^2+c_2n$ and c_3n
 - for sufficiently large of value, c_3n is faster than $c_1n^2+c_2n$
 - for small values of n , either could be faster
 - $c_1=1, c_2=2, c_3=100 \rightarrow c_1n^2+c_2n \leq c_3n$ for $n \leq 98$
 - $c_1=1, c_2=2, c_3=1000 \rightarrow c_1n^2+c_2n \leq c_3n$ for $n \leq 998$
 - break even point
 - no matter what the values of c_1, c_2 , and c_3 , the n beyond which c_3n is always faster than $c_1n^2+c_2n$

- 
- $O(1)$: constant
 - $O(n)$: linear
 - $O(n^2)$: quadratic
 - $O(n^3)$: cubic
 - $O(2^n)$: exponential
 - $O(\log n)$
 - $O(n \log n)$

*Figure 1.7:Function values

		Instance characteristic n						
Time	Name	1	2	4	8	16	32	
1	Constant	1	1	1	1	1	1	
$\log n$	Logarithmic	0	1	2	3	4	5	
n	Linear	1	2	4	8	16	32	
$n \log n$	Log linear	0	2	8	24	64	160	
n^2	Quadratic	1	4	16	64	256	1024	
n^3	Cubic	1	8	64	512	4096	32768	
2^n	Exponential	2	4	16	256	65536	4294967296	
$n!$	Factorial	1	2	24	40326	20922789888000	26313×10^{33}	

Figure 1.7 Function values

*Figure 1.8: Plot of function values

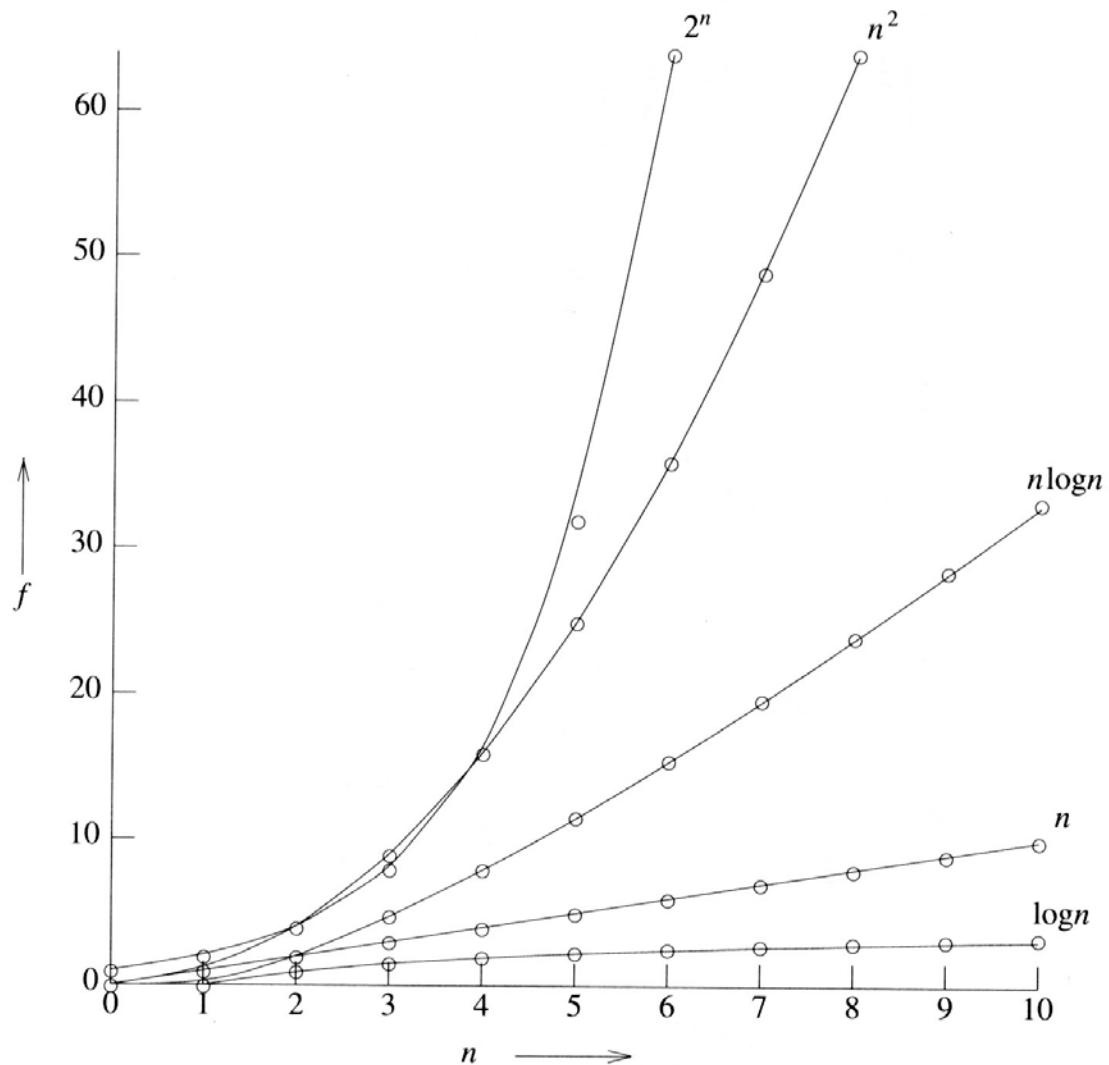


Figure 1.8 Plot of function values

***Figure 1.9:** Times on a 1 billion instruction per second computer

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