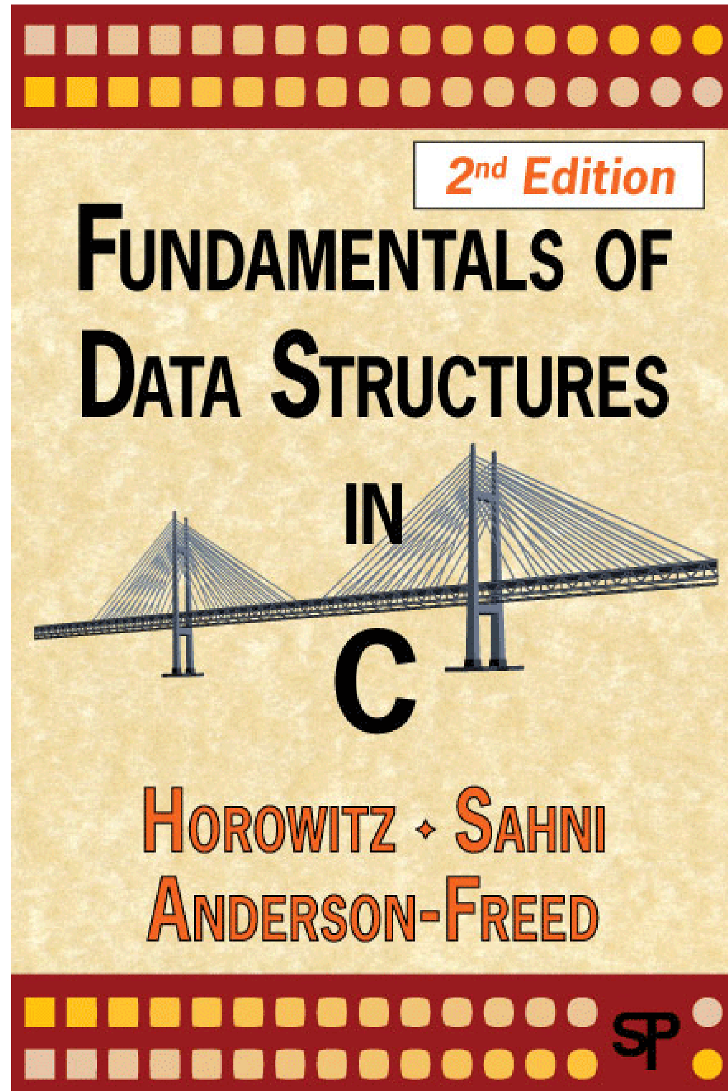
# Data Structures

# Books

Fundamentals of Data Structures in C, 2nd Edition.

(開發圖書，(02) 8242-3988)



# Administration

- Instructor:
  - 曾學文
  - Office: Room 908
  - Email: hwtseng@nchu.edu.tw
  - Tel: 04-22840497 ext. 908
- Office Hours:
  - (Wednesday)13:30~15:00;
- Grade:
  - Quiz 10%
  - Homework 20%
  - Midterm Exam 20%

# Introductory

- Raise your hand is always welcome!
- Slides are not enough. To master the materials, page-by-page reading is necessary.
- No phone, walk, sleep, and late during the lecture time.
- Data structure is not the fundamental course for programming.
- Do not copy the homework.

# Outline

- Basic Concept
- Arrays and Structures
- Stacks and Queues
- Lists
- Trees

# 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",

# 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 .
  - 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 $\in$ *Nat_Number*; *TRUE*, *FALSE* $\in$ *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*

**::= is defined as**

# 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;
}
```

$S_{sum}(I) = 0$

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;
 }
```

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

---

Assumptions:

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

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

# Time Complexity

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

- C: Compile time
  independent of instance characteristics
- $T_P$ : Run (execution) time
- Definition $\boxed{T_P(n)=c_a ADD(n)+c_s SUB(n)+c_l LDA(n)+c_{st} STA(n)}$
  A *program step* is a syntactically or semantically meaningful program segment whose execution time is <u>independent of the instance characteristics</u>.
- Example
  - abc = a + b + b * c + (a + b - c) / (a + b) + 4.0
  - abc = a + b + c

Regard as the same unit
and machine independent

# Methods to compute the step count

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

# 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 |

# 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 1.13

initial
n = 3,     count = 0

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

count = 1

# Program 1.13

initial
n = 3,     count = 0

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

i = 0, count = 2

# Program 1.13

initial
n = 3,    count = 0

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

count = 3

# Program 1.13

initial
n = 3,     count = 0

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

i = 1, count = 4

# Program 1.13

initial
n = 3,     count = 0

```
float sum(float list[], int n) {
    float tempsum = 0;
    int i;
    for(i = 0; i < n; i++){
        tempsum += list[i];              count = 5
    }
    return tempsum;
}
```

# Program 1.13

initial
n = 3,    count = 0

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

i = 2, count = 6

# Program 1.13

initial
n = 3,    count = 0

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

count = 7

# Program 1.13

initial
n = 3,  count = 0

```
float sum(float list[], int n) {
    float tempsum = 0;
    int i;
    for(i = 0; i < n; i++){          i = 3, count = 8
        tempsum += list[i];
    }
    return tempsum;
}
```

# Program 1.13

initial
n = 3,    count = 0

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

count = 9

回outline

# Program 1.13

initial
n = 3,    count = 0

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

1次

n+1次

n次

\+ 1次

2n+3次

***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

# Program 1.15

initial

n = 3,     count = 0

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

n=3, count = 1

# Program 1.15

initial
n = 3,    count = 0

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

n=3, count = 2

# Program 1.15

initial
n = 3,     count = 0

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

n=2, count = 3

# Program 1.15

initial
n = 3,     count = 0

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

n=2, count = 4

# Program 1.15

initial
n = 3,     count = 0

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

n=1, count = 5

# Program 1.15

initial
n = 3,    count = 0

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

n=1, count = 6

# Program 1.15

initial
n = 3,    count = 0

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

n=0, count = 7

# Program 1.15

initial
n = 3,  count = 0

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

n=0, count = 8

# Program 1.15

initial

n = 3,     count = 0

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

n+1次

n次

\+ 1次

---

2n+2次

# 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

***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];                rows * cols
 }
```

# 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 |

***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 */
}
```

2*rows * cols + 2 rows  + 1

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
             c[][MAX_SIZE], int rows, int cols) {
    int i, j;
    for(i = 0;i < rows; i++){                  i = 0, count = 1
        for(j = 0;j < cols; j++){
            c[i][j] = a[i][j] + b[i][j];
        }
    }
}
```

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
              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];
        }
    }
}
```

j = 0, count = 2

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
              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];
        }
    }
}
```

count = 3

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
            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];
      }
   }
}
```

j = 1, count = 4

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
            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];
      }
   }
}
```

count = 5

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
              c[][MAX_SIZE], int rows, int cols) {
    int i, j;
    for(i = 0;i < rows; i++){
        for(j = 0;j < cols; j++){                    j = 2, count = 6
            c[i][j] = a[i][j] + b[i][j];
        }
    }
}
```

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
              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];
        }
    }
}
```

count = 7

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
            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];
        }
    }
}
```

j = 3, count = 8

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
              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];
        }
    }
}
```

i = 1, count = 9

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
             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];
        }
    }
}
```

j = 0, count = 10

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
              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];
        }
    }
}
```

count = 11

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
              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];
        }
    }
}
```

j = 1, count = 12

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
              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];
        }
    }
}
```

count = 13

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
              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];
        }
    }
}
```

j = 2, count = 14

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
           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];
    }
  }
}
```

count = 15

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
             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];
        }
    }
}
```

j = 3, count = 16

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
             c[][MAX_SIZE], int rows, int cols) {
    int i, j;
    for(i = 0;i < rows; i++){                i = 2, count = 17
        for(j = 0;j < cols; j++){
            c[i][j] = a[i][j] + b[i][j];
        }
    }
}
```

回outline

# Program 1.17

initial
rows=2, cols=3, count = 0

```
void add(int a[][MAX_SIZE], b[][MAX_SIZE],
            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];
        }
    }
}
```

rows+1 次

rows*(cols+1) 次

+ rows*cols 次

2rows*cols+2rows+1 次

# 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++) {           /* row +1*/
      for (j = 0; j < cols; j++)         /*  row * (col +1) */
          printf(“%d”,  matrix[i][j]);   / row * col */
      printf( “\n”);                     /* row */
   }
}
```

2*row*col + 2 row + row +1

# Asymptotic Notation

## Definition

- **Big-Oh (O)**

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

- **Big-Omega (Ω)**

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

- **Big-Theta (Θ)**

  $f(n) = \Theta(g(n))$ iff there exist positive constants $c_1, c_2$ and $n_0$ such that $c_1 g(n) \le f(n) \le c_2 g(n)$, for all $n, n \ge n_0$.

# Asymptotic Notation (O)

- Definition
  $f(n) = O(g(n))$ iff there exist positive constants c and $n_0$ such that $f(n) \le cg(n)$ for all n, $n \ge n_0$.
- Examples
  - $3n+2=O(n)$ /* $3n+2 \le 4n$ for $n \ge 2$ */
  - $3n+3=O(n)$ /* $3n+3 \le 4n$ for $n \ge 3$ */
  - $100n+6=O(n)$ /* $100n+6 \le 101n$ for $n \ge 6$ */
  - $10n^2+4n+2=O(n^2)$ /* $10n^2+4n+2 \le 11n^2$ for $n \ge 5$ */
  - $6*2^n+n^2=O(2^n)$ /* $6*2^n+n^2 \le 7*2^n$ for $n \ge 4$ */

# Asymptotic Notation (Θ)

- Definition

  $f(n) = \Theta(g(n))$ iff there exist positive constants $c_1, c_2$ and $n_0$ such that $c_1 g(n) \le f(n) \le c_2 g(n)$, for all $n, n \ge n_0$.

- Examples
  - $3n + 2 = \Theta(n)$

    $3n + 2 \ge 3n$ for all $n \ge 2$ and $3n + 2 \le 4n$ for all $n \ge 2$,
    so $c_1 = 3, c_2 = 4$ and $n_0 = 2$
  - $10n^2 + 4n + 2 = \Theta(n^2)$

    $10n^2 + 4n + 2 \ge 10n^2$ for all $n \ge 5$ and $10n^2 + 4n + 2 \le 11n^2$ for all $n \ge 5$,
    so $c_1 = 10, c_2 = 11$ and $n_0 = 5$
  - $6 * 2^n + n^2 = \Theta(2^n)$

    $6 * 2^n + n^2 \ge 6 * 2^n$ for all $n \ge 4$ and $6 * 2^n + n^2 \le 7 * 2^n$ for all $n \ge 4$,
    so $c_1 = 6, c_2 = 7$ and $n_0 = 4$

# 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
    - $c_1=1, c_2=2, c_3=100$ --> $c_1n^2+c_2n \leq c_3n$ for $n \leq 98$
    - $c_1=1, c_2=2, c_3=1000$ --> $c_1n^2+c_2n \leq c_3n$ for $n \leq 998$
  - break even point
    - no matter what the values of c1, c2, and c3, 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 | $26313 \times 10^{33}$ |

**Figure 1.7** Function values

# *Figure 1.8:Plot of function values

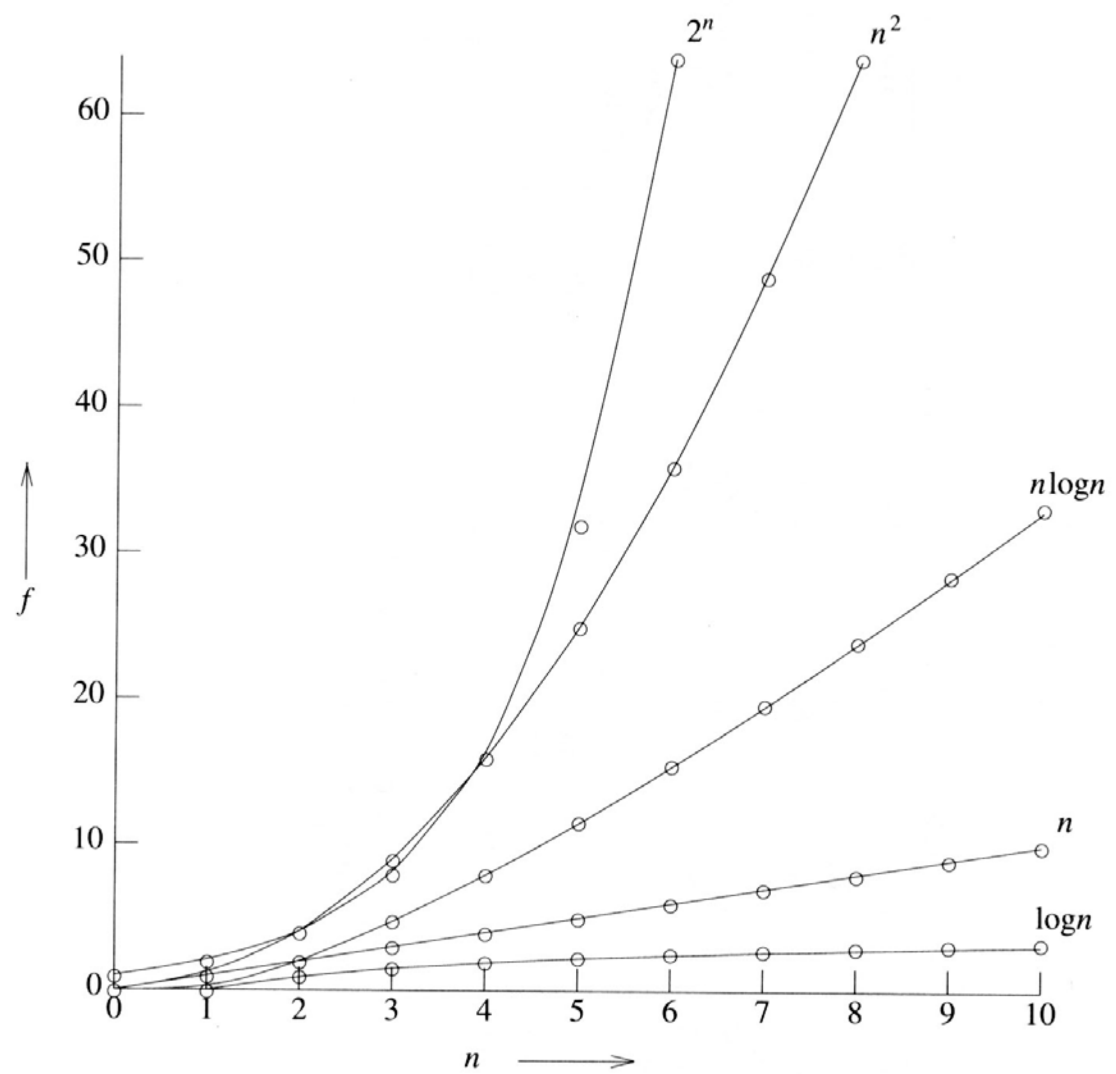



**Figure 1.8** Plot of function values

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

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