Data Structures

Chapter 1 Basic Concepts
<table>
<thead>
<tr>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview : System Life Cycle</td>
</tr>
<tr>
<td>Pointers and Dynamic Memory Allocation</td>
</tr>
<tr>
<td>Algorithm Specification</td>
</tr>
<tr>
<td>Data Abstraction</td>
</tr>
<tr>
<td>Performance Analysis</td>
</tr>
<tr>
<td>Performance Measurement</td>
</tr>
</tbody>
</table>
System Life Cycles

1. Requirements: a set of spec. (input ? output ?)
2. Analysis:
   - **bottom-up**: Akin to constructing a building from a generic blueprint
   - **top-down**: Divide program into manageable segments
3. Design: language independent
   - **data objects**: creating abstract data types (ADT)
   - **operations**: specifying algorithms
4. Refinement and Coding
5. Verification
   - Correctness proofs
     - proving in mathematics
     - time-consuming
   - Testing
     - Including all possible scenarios
     - Correction vs. performance
   - Error removal (debugging)
Pointers in C/C++

- The two most important operators used with the pointer type are:
  - & the address operator
  - * the dereferencing (or indirection) operator

- Example:
  ```c
  int i, *pi;
  pi = &i;
  
  &i returns the address of i and assigns it as the value of pi
  
  To assign a value to i we can say:
  i = 10; or
  *pi = 10;
  ```
Dynamic Memory Allocation

- We may not know how much space we will need, nor do we wish to allocate some vary large area that may never be required
- C provides a mechanism, called a heap, for allocating storage at run-time
  - We can call a function, malloc, to request the amount of memory.
  - Later we can call another function, free, to return the area of memory to the system
- C++ uses the function, new, to request the amount of memory and uses the function, delete, to release the memory
Program 1.1: Allocation and deallocation of memory

C:

```c
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\n", *pi);
printf("a float = %f\n", *pf);
free(pi);
free(pf);
```

C++:

```c
int i, *pi;
float f, *pf;
pi = new int;
pf = new float;
*pi = 1024;
*pf = 3.14;
cout << "an integer= " << *pi << endl;
cout << "a float= " << *pf << endl;
delete pi;
delete pf;
```

Demo: ~/data_structure/chap1/memory_allocation.c (cc)

telnet ntut.edu.tw
> joe test.c (edit)
> gcc test.c (compile)
> a.out (execute)
Robust Version of Allocating Memory

- Check if lack of sufficient memory
  
  ```c
  if ((pi = (int *) malloc(sizeof(int))) == NULL ||
    (pf = (float *) malloc(sizeof(float))) == NULL) {
    fprintf(stderr, “Insufficient memory”);
    exit(EXIT_FAILURE);
  }
  ```

- Macro version
  
  ```c
  #define MALLOC(p, s) 
  if ( !((p) = malloc(s))) { 
    fprintf(stderr, “Insufficient memory”);
    exit(EXIT_FAILURE);
  }
  ```

  Ex.
  
  MALLOC(pi, sizeof(int));
  MALLOC(pf, size(float));

  Demo: ~/data_structure/chap1/macro.c
Pointers Can Be Dangerous

- Set all pointers to NULL when they are not actually pointing to an object
- Use explicit type cast when converting between pointer types
  - Example:
    ```c
    pi = malloc(sizeof(int)); /* assign to pi a pointer to int */
    pf = (float *) pi;       /* casts an int pointer to a float pointer */
    ```
- Define explicit return types for functions.
  - Pointers may have the same size as type int. The function return type defaults to int which can later be interpreted as a pointer
Algorithm

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

- **Criteria**
  1. **Input**: zero or more data
  2. **Output**: at least one
  3. **Definiteness**: each instruction is clear and unambiguous
  4. **Finiteness**: terminate after a *finite* number of steps
     - Program may not terminate
  5. **Effectiveness**: instruction is basic enough to be carried out 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.
Describing an Algorithm

- Translating a problem into an algorithm:
  - Natural language
    - Instructions must be definite
  - Graphic representation (Flowchart)
    - work well only if the algorithm is small and simple
  - Combining English and C (Pseudo code)
Example 1.1: Selection Sort

- Problem: Sort a collection of $n \geq 1$ integers.

- Natural Language: *from those integers that are currently unsorted, find the smallest and place it next in the sorted list*
Example 1.1: Selection Sort (cont.)

- Pseudo code: English & C

```c
for (i = 0; i < n - 1; 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

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

- Two subtasks:
  - finding the smallest integer
  - interchanging it with list[i] ➔ swap
Example 1.1: Selection Sort (cont.)

- **Swap**
  - **Macro**
    
    ```
    #define SWAP(x, y, t)   ((t) = (x), (x) = (y), (y) = (t))
    ```
  - **Function**
    ```
    void swap(int *x, int *y) {
        int temp = *x;
        *x = *y;
        *y = temp;
    }
    ```
  - **Example:**
    ```
    int a=3, b=5, tmp;
    swap(&a, &b);
    SWAP(a, b, tmp);
    ```
Example 1.1: Selection Sort (cont.)

- Program 1.4: C code for 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(1);
    }
    for (i = 0; i < n; i++) { /*randomly generate numbers*/
        list[i] = rand() % 1000;
        printf("%d ",list[i]);
    }
    sort(list,n);
    printf("\nSorted array:\n");
    for (i = 0; i < n; i++) /* print out sorted numbers */
        printf("%d ",list[i]);
    printf("\n");
}

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);
    }
}
```

Demo:
~/data_structure/chap1/selection_sort.c
Example 1.1: Selection Sort (cont.)

- **Theorem**: Function sort(list, n) correctly sorts a set of \( n \geq 1 \) distinct integers.
- **Proof**: (by math. induction)
  - when \( i=q \), we have \( list(q) < list(r) \), for \( q < r < n \)
  - when \( i > q \), list[0] through list[q] unchanged
  - The last iteration \( (i=n-2) \), list[0] \( \leq \) list[1] \( \leq \ldots \leq \) list[n-1]
Example 1.2: Binary Search

- Problem:
  Given a sorted list, figure out if an integer `searchnum` is in it.
  If yes, return its index `i`, such that `list[i] = searchnum`. Otherwise, return -1.
Example 1.2: Binary Search (cont.)

- Program 1.5: Searching a sorted list
  
  ```
  while (there are more integers to check) {
      middle = (left + right) / 2;
      if (searchnum < list[middle])
          right = middle - 1;
      else if (searchnum == list[middle])
          return middle;
      else left = middle + 1;
  }
  ```

- Example 1.2 [Binary search]:

<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
<th>middle</th>
<th>list[middle]</th>
<th>searchnum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
<td>32</td>
<td>&lt; 47</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>5</td>
<td>50</td>
<td>&gt; 47</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>47</td>
<td>== 47</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
<td>32</td>
<td>&gt; 16</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>14</td>
<td>&lt; 16</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>28</td>
<td>&gt; 16</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example 1.2: Binary Search (cont.)

- Iterative implementation (Program 1.7)

```c
#define COMPARE(x,y)  (((x) < (y)) ? -1: ((x) == (y)) ? 0: 1)
int binsearch(int list[], int searchnum, int left, int right)
{
    int middle;
    while (left <= right) {
        middle = (left + right) /2;
        switch (COMPARE(list[middle], searchnum)) {
            case -1: left = middle + 1;      /* list[middle] < searchnum */
                break;
            case 0 : return middle;        /* list[middle] = searchnum */
            case 1 : right = middle - 1;    /* list[middle] > searchnum */
        }
    }
    return -1;
}
```

Demo: ~/data_structure/chap1/binary_search.c
Recursive Algorithms

- Direct recursion
  - Functions can call themselves

- Indirect recursion
  - Functions can call other functions that invoke the calling function again

- How to determine that express an algorithm recursively?
  - The problem itself is defined recursively
    - ex.: factorials, Fibonacci numbers…
  - Any function using assignment, *if-else* and *while* statements can be written recursively
Recursive Algorithms (cont.)

- $N! = N \times (N-1)!$

```c
#include <stdio.h>

int fact(int n) {
    if (n==0) return 1; else return(n * fact(n-1));
}

int main( )
{
    printf("5!=%d\n",fact(5));
}
```

```c
int fact(int n)
{
    if (n==0) return 1;
    else return(n * fact(n-1));
}
```
Recursive Algorithms (cont.)

```c
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

Demo: ~/data_structure/chap1/binary_search_recursive.c
Example 1.4: Permutations

- Problem:
  Given a set of \( n \geq 1 \) elements, print out all possible permutations of this set.
  - There are \( n! \) permutations
- All possible permutations of \( \{a, b, c, d\} \) (\( n=4 \))
  - \( a \) followed by all permutations of \( \{b, c, d\} \)
    - \( b \) followed by all permutations of \( \{c, d\} \)
      - \( c, d \), \( d, c \) → \( b, c, d \), \( b, d, c \)
    - \( c \) followed by all permutations of \( \{b, d\} \)
      - \( b, d \), \( d, b \) → \( c, b, d \), \( c, d, b \)
    - \( d \) followed by all permutations of \( \{c, b\} \)
      - \( c, b \), \( b, c \) → \( d, c, b \), \( d, b, c \)
  - \( b \) followed by all permutations of \( \{a, c, d\} \)
    - \( b, a, c, d \), \( b, a, d, c \), \( b, c, a, d \), \( b, c, d, a \), \( b, d, a, c \), \( b, d, c, a \)
  - \( c \) followed by all permutations of \( \{a, b, d\} \)
    - \( c, a, b, d \), \( c, a, d, b \), \( c, b, a, d \), \( c, b, d, a \), \( c, d, a, b \), \( c, d, b, a \)
  - \( d \) followed by all permutations of \( \{a, b, c\} \)
    - \( d, a, b, c \), \( d, a, c, b \), \( d, b, a, c \), \( d, b, c, a \), \( d, c, a, b \), \( d, c, b, a \)
Example 1.4: Permutations (cont.)

```c
void perm(char *list, int i, int n)
    /* generate all the permutations of list[i] to list[n] */
    {
        int j, temp;
        if (i == n) {
            for (j = 0; j <= n; j++)
                printf("%c", list[j]);
            printf(" ");
        }
        else {
            /* list[i] to list[n] has more than one permutation,
egenerate these recursively */
            for (j = i; j <= n; j++) {
                SWAP(list[i], list[j], temp);
                perm(list, i+1, n);
                SWAP(list[i], list[j], temp);
            }
        }
    }
```

Program 1.9: Recursive permutation generator

Demo: ~/data_structure/chap1/permutation.c
Example 1.4: Permutations (cont.)

Example:

\[ \text{n=3; } A = \{a, b, c\}; \text{ } a \text{ followed by all permutations of } \{b, c\} \]

Call \( \text{PERM}(A,1,3) \)

\[ \text{PERM}(A,1,3) \]

\( i \neq n \)

\( j=1,2,3 \)

\( j=1 \)

\( A = \{a,b,c\} \)

\( j=2 \)

\( A = \{b,a,c\} \)

\( j=3 \)

\( A = \{c,b,a\} \)

end. stop

\[ \text{PERM}(A,2,3) \]

\( i \neq n \)

\( j=2,3 \)

\( j=2 \)

\( A = \{a,b,c\} \)

\( j=3 \)

\( A = \{a,c,b\} \)

end. stop

\[ \text{PERM}(A,3,3) \]

\( i=n \)

\( i=n \)

\( a \quad b \quad c \)

\( a \quad c \quad b \)
Data Abstraction

- **Data Type**
  A **data type** is a collection of **objects** and a set of **operations** that act on those objects.
  - For example, the data type **int** consists of the objects \{0, +1, -1, +2, -2, …, INT_MAX, INT_MIN\} and the operations +, -, *, /, and %.

- **The data types of C**
  - **basic types** : char, int, float, double
    - may be modified by short, long, and unsigned
  - **grouping data** : array, struct
  - **pointer**
Abstract Data Type (ADT)

- Abstract Data Type
  - An abstract data type (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.
    ➔ Implementation-independent: we know what is does, but not necessarily how it will do it

- Operation specification
  - function name
  - the types of arguments
  - the type of the results
Abstract Data Type (cont.)

**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* is *NaturalNumber*; *TRUE, FALSE* is *Boolean* and where +, -, <, and == are the usual integer operations

```
NaturalNumber Zero() ::= 0
Boolean IsZero(x) ::= if (x) return FALSE
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*
Performance Analysis

Criteria for Judging a Program

1. Does the program meet the original specifications of the task?
2. Does it work correctly?
3. Does the program contain documentation that shows how to use it and how it works?
4. Does the program effectively use functions to create logical units?
5. Is the program’s code readable?
6. Does the program efficiently use primary and secondary storage?
7. Is the program’s running time acceptable for the task?
Performance Analysis (cont.)

- Performance Analysis (*machine independent*)
  - space complexity: storage requirement
  - time complexity: computing time

- Performance Measurement (*machine dependent*)
  - obtains machine-dependent running time
Space Complexity

- The space complexity of a program is the amount of memory that it needs to run to completion.
- Space complexity \( S(P) = c + S_P(I) \)
  - Fixed Space Requirements (\( c \)): constant
    - Independent of the characteristics of the inputs and outputs
    - Instruction space, space for simple variables, fixed-size structured variable, constants
    - Prepared by the compiler in compile-time
  - 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
    - Needed in run-time
Space Complexity (cont.)

- Example

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

Program 1.10: Simple arithmetic function

\[ S_{abc}(I) = 0 \]
Space Complexity (cont.)

- Example

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

Program 1.11: Iterative function for summing a list of numbers

- language dependent:
  - C → $S_{sum}(I) = S_{sum}(n) = 0$
    - C passes an array by the address of the first element of the array
  - Pascal → $S_{sum}(I) = S_{sum}(n) = n$
    - Pascal passes an array by all its value
Space Complexity (cont.)

- Example:

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

Program 1.14: **Recursive function** for summing a list of numbers

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Number of bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter: array pointer</td>
<td>list[]</td>
<td>4</td>
</tr>
<tr>
<td>parameter: integer</td>
<td>n</td>
<td>4</td>
</tr>
<tr>
<td>return address: (used internally)</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>TOTAL (per recursive call)</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

\[ S_{rsum}(I) = S_{rsum}(n) = 12n \]
Time Complexity

- The time complexity of a program is the amount of computer time that it needs to run to completion.

\[ T(P) = C + T_P(I) \]

- The time, \( T(P) \), taken by a program, \( P \), is the sum of its compile time \( C \) and its run (or execution) time, \( T_P(I) \).

- Fixed time requirements: Compile time \( (C) \), independent of instance characteristics.

- Variable time requirements: Run (execution) time \( T_P \)
  - analyzing object code (compiler-dependent)
    - Example: \( T_P(n) = c_a \cdot ADD(n) + c_s \cdot SUB(n) + c_l \cdot LDA(n) + c_{st} \cdot STA(n) \)
      - \( c_a \): time needed to perform ADD operation
      - \( ADD(n) \): the number of additions that are performed when the program is run with instance characteristic \( n \)
  - measuring with system clock
  - counting program steps (machine-independent)
A program step is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

Example

- $a = 2$
- $a = 2b + 3c/d - e + f/g/a/b/c$

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 $(s/e) \times$ Frequency
  - add up the contribution of all statements
Example: Iterative summing of a list of numbers

- Program 1.13: Adding 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 */
    return tempsum; count++; /* for return */
  }

- Program 1.14: Simplified version of Program 1.13
  float sum(float list[], int n) {
    float tempsum = 0;
    for (int i = 0; i < n; i++)
      count += 2;
    count += 3;
    return 0;
  }

\[ T(P) = 2n + 3 \]
Example: Recursive summing of a list of numbers

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

\[
T(0) = 2 \\
T(n) = 2 + T(n-1) \\
T(n) = 2+T(n-1) \\
\quad = 2 + 2 + T(n-2) \\
\quad = 2 + 2 + 2 + T(n-3) \\
\quad \vdots \\
\quad = 2n + T(0) \\
\quad = 2n + 2
\]

\[T(P) = 2n+2\]
# Data Structures Basic Concepts

## Example: Matrix addition

```c
#define MAX_SIZE 3
void main()
{
    int a[2][3]={{1,2,3},{4,5,6}};
    int b[2][3]={{2,4,6},{1,3,5}};
    int c[2][3];
    int i,j;
    add(a, b, c, 2, 3);
    for (i=0 ; i < 2; i++){
        for (j=0; j < 3; j++){
            printf(" a[%d][%d]=%d\n",i,j,a[i][j]);
            printf("+ b[%d][%d]=%d\n",i,j,b[i][j]);
            printf("= c[%d][%d]=%d\n",i,j,c[i][j]);
            printf("\n");
        }
    }
}
```

 Demo: ~/data_structure/chap1/matrix_add.c

- Program 1.16: Matrix addition

```c
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];
}
```
Example: Matrix addition (cont.)

- Program 1.18: Matrix addition with count statements

```c
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++) {
        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 */
}
```

Example: Matrix addition (cont.)

- Program 1.18: Simplification of Program 1.17

```c
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 += 2;
}
```

\[ T(P) = 2 \cdot \text{rows} \cdot \text{cols} + 2 \cdot \text{rows} + 1 \]
Tabular Method

- Determine the total number of steps contributed by each statement **Step per Execution (s/e) \times Frequency**
- Add up the contribution of all statements
- Example 1.12: Iterative function to sum a list of numbers

<table>
<thead>
<tr>
<th>Statement</th>
<th>s/e</th>
<th>Frequency</th>
<th>Total step</th>
</tr>
</thead>
<tbody>
<tr>
<td>float sum( float list[ ], int n )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>float tempsum = 0;</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>int i;</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>for ( i = 0; i &lt; n; i++ )</td>
<td>1</td>
<td>n+1</td>
<td>n+1</td>
</tr>
<tr>
<td>tempsum += list[i];</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>return tempsum;</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td><strong>2n+3</strong></td>
</tr>
</tbody>
</table>
Example 1.13: Recursive function to sum a list of numbers

<table>
<thead>
<tr>
<th>Statement</th>
<th>s/e</th>
<th>Frequency</th>
<th>Total step</th>
</tr>
</thead>
<tbody>
<tr>
<td>float rsum( float list[ ], int n )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>{</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>if (n)</td>
<td>1</td>
<td>n+1</td>
<td>n+1</td>
</tr>
<tr>
<td>return rsum(list, n-1)+ list[n-1];</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>return 0;</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>2n+2</td>
</tr>
</tbody>
</table>
Example 1.14: Matrix addition

<table>
<thead>
<tr>
<th>Statement</th>
<th>s/e</th>
<th>Frequency</th>
<th>Total step</th>
</tr>
</thead>
<tbody>
<tr>
<td>void add( int a[ ][MAX_SIZE] ... ) { int i, j; for ( i = 0; i &lt; rows; i++ ) for ( j = 0; j &lt; cols; j++ ) c[i][j] = a[i][j] + b[i][j]; }</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>rows+1</td>
<td>rows+1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>rows · (cols+1)</td>
<td>rows · cols + rows</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>rows · cols</td>
<td>rows · cols</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Total \(2 \text{rows} \cdot \text{cols} + 2 \text{rows} + 1\)
Time Complexity of binsearch()

- Three kinds of step counts
  - best case: searchnum = 32 → step counts = 1
  - worst case: searchnum = 16 → step counts = 4
  - average case: searchnum = 14 → step counts = 2

<table>
<thead>
<tr>
<th></th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
<th>[5]</th>
<th>[6]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>9</td>
<td>14</td>
<td>28</td>
<td>32</td>
<td>47</td>
<td>50</td>
<td>55</td>
</tr>
</tbody>
</table>
Asymptotic Notation (Ο, Ω, Θ)

- Motivation
  - Determining exact step count is difficult.
  - The notion of step is itself inexact.
    - ex. Both \( x=y \) and \( x=y+z+(x/y)+(x*y*z-x/z) \) count as one step

- Approach
  - Complexity of \( c_1 n^2 + c_2 n \) and \( c_3 n \)
    - for sufficiently large values of \( n \), \( c_3 n \) is faster than \( c_1 n^2 + c_2 n \)
    - for small values of \( n \), either could be faster
      - \( c_1=1, c_2=2, c_3=100 \Rightarrow c_1 n^2 + c_2 n \leq c_3 n \) for \( n \leq 98 \)
      - \( c_1=1, c_2=2, c_3=1000 \Rightarrow c_1 n^2 + c_2 n \leq c_3 n \) for \( n \leq 998 \)
  - break even point (98, 998)
    - no matter what the values of \( c_1, c_2, \) and \( c_3 \), the \( n \) beyond which \( c_3 n \) is always faster than \( c_1 n^2 + c_2 n \)
  - there is little advantage in determining the exact values of \( c_1, c_2, \) and \( c_3 \)
O: Big “oh”

- 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
  - \( 3n+2=O(n) \) /* \( 3n+2 \leq 3n+n \leq 4n \) 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 \( n \geq 4 \) */

- \( f(n) = O(g(n)) \) means that \( g(n) \) is an upper bound of \( f(n) \)
  - \( 10n^2 + 4n + 2 = O(n^4) \) as \( 10n^2 + 4n + 2 \leq 10n^4 \) for all \( n \geq 2 \)
  - \( g(n) \) should be as small a function of \( n \) as one can come up with for which \( f(n) = O(g(n)) \)

- Theorem 1.2: If \( f(n) = a_mn^m + \ldots + a_1n + a_0 \), then \( f(n) = O(n^m) \)

Proof: \( f(n) \leq \sum_{i=0}^{m} |a_i|n^i \leq n^m \sum_{i=0}^{m} |a_i|n^{i-m} \leq n^m \sum_{i=0}^{m} |a_i|, \) for all \( n \geq 1 \)
**Ω : Omega**

- **Definition**
  
  \[ f(n) = \Omega (g(n)) \] iff there exist positive constants \( c \) and \( n_0 \) such that
  
  \[ f(n) \geq cg(n) \] for all \( n, n \geq n_0 \)

- **Ex-1.16 :**
  
  - \( 3n+3 = \Omega(n) \) /* \( 3n+3 \geq 3n \) for \( n \geq 1 \) */
  - \( 3n+3 = \Omega (1) \) /* \( 3n+3 \geq 1 \) for \( n \geq 1 \) */
  - \( 3n+3 \neq \Omega (n^2) \)
  - \( 10n^2+4n+2 = \Omega (n^2) \) /* \( 10n^2+4n+2 \geq n^2 \) for \( n \geq 1 \) */
  - \( 6*2^n+ n^2 = \Omega (2^n) \) /* \( 6*2^n+ n^2 \geq 2^n \) for \( n \geq 1 \) */
  - \( 6*2^n+ n^2 = \Omega (n^2) \)

- \( f(n) = \Omega (g(n)) \) means that \( g(n) \) is an lower bound of \( f(n) \)
  
  - \( g(n) \) should be as large a function of \( n \) as possible for which the statement \( f(n) = \Omega (g(n)) \) is true

- **Theorem 1.3 :** If \( f(n) = a_mn^m + \ldots + a_1n + a_0 \) and \( a^m > 0 \), then \( f(n) = \Omega (n^m) \)
Θ : Theta

- **Definition**
  \[ f(n) = \Theta(g(n)) \iff \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that} \]
  \[ c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n, n \geq n_0 \]

- **Ex-1.17 :**
  - \( 3n + 2 = \Theta(n) \) /* \( 3n + 2 \geq 3n \text{ for all } n \geq 2 \) and \( 3n + 2 \leq 4n \text{ for all } n \geq 2 */
  - \( 10n^2 + 4n + 2 = \Theta(n^2) \)
  - \( 6 \cdot 2^n + n^2 = \Theta(2^n) \)
  - \( 3n + 2 \neq \Theta(1) \)
  - \( 6 \cdot 2^n + n^2 \neq \Theta(n^2) \)

- **Theorem 1.4:** If \( f(n) = a_m n^m + ... + a_1 n + a_0 \), and \( a^m > 0 \), then \( f(n) = \Theta(n^m) \).

- The complexity in terms of \( O \), \( \Omega \), and \( \Theta \) can be determined quite easily without determining the exact step count.
Time Complexity of Matrix Addition

- Tabular approach + Asymptotic complexity

<table>
<thead>
<tr>
<th>Statement</th>
<th>Asymptotic complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>void add(int a[][MAX]...)</td>
<td>0</td>
</tr>
<tr>
<td>{</td>
<td>0</td>
</tr>
<tr>
<td>int i, j;</td>
<td>0</td>
</tr>
<tr>
<td>for (i=0; i&lt;rows; i++)</td>
<td>$\Theta(\text{rows})$</td>
</tr>
<tr>
<td>for (j=0; j&lt;cols; j++)</td>
<td>$\Theta(\text{rows} \times \text{cols})$</td>
</tr>
<tr>
<td>c[i][j]=a[i][j]+b[i][j];</td>
<td>$\Theta(\text{rows} \times \text{cols})$</td>
</tr>
<tr>
<td>}</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>$\Theta(\text{rows} \times \text{cols})$</td>
</tr>
</tbody>
</table>
Time complexity of binary search

- each iteration of `while` loop takes $\Theta(1)$ time
- iterated loops of `while`
  - worst case: $\log_2(n + 1)$
  - best case: 1
- complexity of `binsearch()`
  - worst case: $\Theta(\log n)$
  - best case: $\Theta(1)$
Time complexity of permutations

- Permutation: $\Theta(n^n!)$ ← see Program 1.9

```c
void perm(char *list, int i, int n)
/* generate all the permutations of list[i] to list[n] */
{
    int j, temp;
    if (i == n) {
        for (j = 0; j <= n; j++)
            printf("%c", list[j]);
        printf(" ");
    }
    else {
        /* list[i] to list[n] has more than one permutation,
        generate these recursively */
        for (j = i; j <= n; j++) {
            SWAP(list[i],list[j],temp);
            perm(list,i+1,n);
            SWAP(list[i],list[j],temp);
        }
    }
}
```

There are $n!$ permutations to be printed out and each one takes $\Theta(n)$ time.
Example 1.21: Magic square

- A **magic square** is an $n \times n$ matrix of the integers from 1 to $n^2$ such that the sum of each row and column and the two major diagonals is the same.

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>8</th>
<th>1</th>
<th>24</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td></td>
<td>7</td>
<td>5</td>
<td>23</td>
</tr>
<tr>
<td>22</td>
<td>20</td>
<td>13</td>
<td>6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>21</td>
<td>19</td>
<td>12</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>25</td>
<td>18</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.6: Magic square for $n = 5$ and the common sum = 65


#include <stdio.h>
#define MAX_SIZE 15 /* maximum size of square */
void main(void)
/* construct a magic square, iteratively */
{
    int square[MAX_SIZE][MAX_SIZE];
    int i, j, row, column; /* indices */
    int count; /* counter */
    int size; /* Square size */

    printf("Enter the size of the square: ");
    scanf("%d", &size);
    /* check for input errors */
    if (size < 1 || size > MAX_SIZE ) {
        fprintf(stderr, "Error! Size is out of range\n");
        exit(EXIT_FAILURE);
    }
Example 1.21 : Magic square (cont.)

if (!(size % 2)) {
    print(stderr, “Error! Size is even\n”);
    exit(EXIT_FAILURE);
}
for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
        square[i][j] = 0;
square[0][(size-1) / 2] = 1; /* middle of first row */
/* i and j are current position */
i = 0; /* row */
j = (size – 1) / 2; /* column */
for (count = 2; count <= size * size; count++) {
    row = (i–1 < 0) ? (size–1) : (i–1); /* up */
    column = (j–1 < 0) ? (size–1) : (j–1); /* left */
    if (square[row][column]) {
        i = (++i) % size; /* down */
    }
}
Example 1.21: Magic square (cont.)

```c
else {
    i = row;
    j = column;
}
square[i][j] = count;
}
/* output the magic square */
printf("Magic Square of size %d : \n\n", size);
for (i = 0; i < size; i++) {
    for (j = 0; j < size; j++) {
        printf("%5d", square[i][j]);
        printf("\n");
    }
    printf("\n\n");
}
```

Complexity: $\Theta(n^2)$
To get a feel for how the various functions grow with $n$, you are advised to study Figures 1.7 ~1.9 very closely.

<table>
<thead>
<tr>
<th>Time</th>
<th>Name</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constant</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>log $n$</td>
<td>Logarithmic</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$n$</td>
<td>Linear</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>$n \log n$</td>
<td>Log linear</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>160</td>
</tr>
<tr>
<td>$n^2$</td>
<td>Quadratic</td>
<td>1</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>1024</td>
</tr>
<tr>
<td>$n^3$</td>
<td>Cubic</td>
<td>1</td>
<td>8</td>
<td>64</td>
<td>512</td>
<td>4096</td>
<td>32768</td>
</tr>
<tr>
<td>$2^n$</td>
<td>Exponential</td>
<td>2</td>
<td>4</td>
<td>16</td>
<td>256</td>
<td>65536</td>
<td>4294967296</td>
</tr>
<tr>
<td>$n!$</td>
<td>Factorial</td>
<td>1</td>
<td>2</td>
<td>24</td>
<td>40326</td>
<td>20922789888000</td>
<td>26313 x $10^{33}$</td>
</tr>
</tbody>
</table>

**Figure 1.7** Function values
Figure 1.8 Plot of function values
### Practical Complexities (cont.)

1.9 Times on a 1 billion instruction per second computer

<table>
<thead>
<tr>
<th>$n$</th>
<th>$f(n)=n$</th>
<th>$f(n)=n\log_2 n$</th>
<th>$f(n)=n^2$</th>
<th>$f(n)=n^3$</th>
<th>$f(n)=n^4$</th>
<th>$f(n)=n^{10}$</th>
<th>$f(n)=2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.01 µs</td>
<td>0.03 µs</td>
<td>0.1 µs</td>
<td>1 µs</td>
<td>10 µs</td>
<td>10 sec</td>
<td>1 µs</td>
</tr>
<tr>
<td>20</td>
<td>0.02 µs</td>
<td>0.09 µs</td>
<td>0.4 µs</td>
<td>8 µs</td>
<td>160 µs</td>
<td>2.84 hr</td>
<td>1 ms</td>
</tr>
<tr>
<td>30</td>
<td>0.03 µs</td>
<td>0.15 µs</td>
<td>0.9 µs</td>
<td>27 µs</td>
<td>810 µs</td>
<td>6.83 d</td>
<td>1 sec</td>
</tr>
<tr>
<td>40</td>
<td>0.04 µs</td>
<td>0.21 µs</td>
<td>1.6 µs</td>
<td>64 µs</td>
<td>2.56 ms</td>
<td>121.36 d</td>
<td>18.3 min</td>
</tr>
<tr>
<td>50</td>
<td>0.05 µs</td>
<td>0.28 µs</td>
<td>2.5 µs</td>
<td>125 µs</td>
<td>6.25 ms</td>
<td>3.1 yr</td>
<td>13 d</td>
</tr>
<tr>
<td>100</td>
<td>0.10 µs</td>
<td>0.66 µs</td>
<td>10 µs</td>
<td>1 ms</td>
<td>100 ms</td>
<td>3171 yr</td>
<td>$4 \times 10^{13}$ yr</td>
</tr>
<tr>
<td>1,000</td>
<td>1.00 µs</td>
<td>9.96 µs</td>
<td>1 ms</td>
<td>1 sec</td>
<td>16.67 min</td>
<td>$3.17 \times 10^{13}$ yr</td>
<td>$32 \times 10^{28}$ yr</td>
</tr>
<tr>
<td>10,000</td>
<td>10.00 µs</td>
<td>130.03 µs</td>
<td>100 ms</td>
<td>16.67 min</td>
<td>115.7 d</td>
<td>$3.17 \times 10^{23}$ yr</td>
<td>$3.17 \times 10^{33}$ yr</td>
</tr>
<tr>
<td>100,000</td>
<td>100.00 µs</td>
<td>1.66 ms</td>
<td>10 sec</td>
<td>11.57 d</td>
<td>3171 yr</td>
<td>$3.17 \times 10^{33}$ yr</td>
<td>$3.17 \times 10^{43}$ yr</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1.00 ms</td>
<td>19.92 ms</td>
<td>16.67 min</td>
<td>31.71 yr</td>
<td>$3.17 \times 10^7$ yr</td>
<td>$3.17 \times 10^{43}$ yr</td>
<td></td>
</tr>
</tbody>
</table>

$\mu$s = microsecond = $10^{-6}$ seconds
ms = millisecond = $10^{-3}$ seconds
sec = seconds
min = minutes
hr = hours
d = days
yr = years
Performance Measurement

- **Performance Measurement**
  - Compiler time: not concern here
  - Not consider space requirements
  - Count execution (computing) time only

- **C**’s standard library: \[ \texttt{#include <time.h>} \]

<table>
<thead>
<tr>
<th>Method 1</th>
<th>Method 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start timing</td>
<td>start = clock();</td>
</tr>
<tr>
<td>Stop timing</td>
<td>stop = clock();</td>
</tr>
<tr>
<td>Type returned</td>
<td>clock _t</td>
</tr>
<tr>
<td>Result in seconds</td>
<td>duration =</td>
</tr>
<tr>
<td></td>
<td>((double) (stop–start)) /</td>
</tr>
<tr>
<td></td>
<td>\texttt{CLOCKS_PER_SEC};</td>
</tr>
<tr>
<td></td>
<td>duration =</td>
</tr>
<tr>
<td></td>
<td>(double) difftime(stop,start);</td>
</tr>
</tbody>
</table>

**Figure 1.10:** Event timing in C

- 1 second = CLOCK\_PER\_SEC ticks (Ex. 1000000 ticks)
- Method 1 is far more accurate
- Method 2 is easier
Example 1.22 [Worst case performance of the selection function]:

- worst case: \( a[0] \geq a[1] \geq a[2] \ldots \geq a[n-1] \Rightarrow a[0] \leq a[1] \leq a[2] \ldots \leq a[n-1] \)
- Program 1.24: First timing program for selection sort
- Program 1.25: More accurate timing program for selection sort
- Figure 1.12: Worst-case performance of selection sort \( \Rightarrow O(n^2) \)

Figure 1.12: Graph of worst case performance for selection sort
Performance Measurement (cont.)

Program 1.24: First timing program for selection sort

```c
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
    int i, n, step = 10;
    int a[MAX_SIZE];
    double duration;
    clock_t start;

    /* times for n = 0, 10, ..., 100, 200, ..., 1000 */
    printf("n repetitions time\n");
    for (n = 0; n <= 1000; n += step)
        /* get time for size n */

        /* initialize with worst-case data */
        for (i = 0; i < n; i++)
            a[i] = n - i;

        start = clock();
        sort(a, n);
        duration = (double) (clock() - start)
            / CLOCKS_PER_SEC;
        printf("%d %f\n", n, duration);
        if (n -= 100) step = 100;
}
```

Program 1.25: More accurate timing program for selection sort

```c
#include <stdio.h>
#include <time.h>
#include "selectionSort.h"
#define MAX_SIZE 1001
void main(void)
{
    int i, n, step = 10;
    int a[MAX_SIZE];
    double duration;

    /* times for n = 0, 10, ..., 100, 200, ..., 1000 */
    printf("n repetitions time\n");
    for (n = 0; n <= 1000; n += step)
        /* get time for size n */

        /* initialize with worst-case data */
        long repetitions = 0;
        clock_t start = clock();
        do
            repetitions++;

            /* initialize with worst-case data */
            for (i = 0; i < n; i++)
                a[i] = n - i;

            sort(a, n);
        while (clock() - start < 1000);
        /* repeat until enough time has elapsed */

        duration = (double) (clock() - start)
            / CLOCKS_PER_SEC;
        duration /= repetitions;
        printf("%d %d %f\n", n, repetitions, duration);
        if (n == 100) step = 100;
```
Generating Test Data

- How to obtain more accurate measured time?
  - using large data size
  - repeating more times (Program 1.25)

- Worst-case time estimation
  - Is not always easy
  - Generating a suitable large number of random test data, the “maximum event” is the worst-case estimation
    - Random number generator, `rand` function, range 0 to `RAND_MAX`

- Average-case time estimation
  - All cases, usually, is much harder to obtain.
    - ex. Sort, \( n! \) cases
  - Estimating an average time by generating a suitably large number of random test data
Homework Assignment

- Section 1.3: Exercise 8
- Section 1.5.3: Exercise 2(a), 2(b), 6
References Materials

- Textbook: Fundamentals of Data Structures in C (2/e)
- The slides from Prof. Chern-Tang Lin (NCYU)
- The slides from Prof. Chuan-Yi Tang (NTHU)
- The slides from Prof. Yeh-Ching Chung (NTHU)
- The slides from Prof. Hsin-Hsi Chen (NTU).