DATA STRUCTURES USING 'C'

Lecture-03

Data Structures

Basic Concepts

Overview: System Life Cycle
Algorithm Specification
Data Abstraction
Performance Analysis
Performance Measurement

Data Structures

- What is the "Data Structure" ?
  - Ways to represent data
- Why data structure ?
  - To design and implement large-scale computer system
  - Have proven correct algorithms
  - The art of programming
- How to master in data structure ?
  - practice, discuss, and think

System Life Cycle

- Summary
  - R A D R C V
- Requirements
  - What inputs, functions, and outputs
- Analysis
  - Break the problem down into manageable pieces
  - Top-down approach
  - Bottom-up approach

System Life Cycle(Cont.)

- Design
  - Create abstract data types and the algorithm specifications, language independent
- Refinement and Coding
  - Determining data structures and algorithms
- Verification
  - Developing correctness proofs, testing the program, and removing errors

Verification

- Correctness proofs
  - Prove program mathematically
    - time-consuming and difficult to develop for large system
- Testing
  - Verify that every piece of code runs correctly
    - provide data including all possible scenarios
- Error removal
  - Guarantee no new errors generated
- Notes
  - Select a proven correct algorithm is important
  - Initial tests focus on verifying that a program runs correctly, then reduce the running time

Chapter 1 Basic Concepts

- Overview: System Life Cycle
- Algorithm Specification
- Data Abstraction
- Performance Analysis
- Performance Measurement

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.

Describing Algorithms

- Natural language
  - English, Chinese
    - Instructions must be definite and effectiveness
- Graphic representation
  - Flowchart
    - work well only if the algorithm is small and simple
- Pseudo language
  - Readable
  - Instructions must be definite and effectiveness
- ***Combining English and C++***
  - In this text

Translating a Problem into an Algorithm

- Problem
  - Devise a program that sorts a set of $n >= 1$ integers
- Step I - Concept
  - From those integers that are currently unsorted, find the smallest and place it next in the sorted list
- Step II - Algorithm
  - *for (i= 0; i< n; i++)*{

    Examine *list[i]* to *list[n-1]* and suppose that the smallest integer is *list[min]*;

    Interchange *list[i]* and *list[min]*;

    }

Translating a Problem into an Algorithm(Cont.)

- Step III - Coding

```
void sort(int *a, int n)
{
    for (i= 0; i< n; i++)
    {
    int j= i;
    for (int k= i+1; k< n; k++){
        if (a[k ]< a[ j]) j= k;
    int temp=a[i]; a[i]=a[ j]; a[ j]=temp;
   }
}
```

Correctness Proof

- Theorem
  - Function *sort(a, n)* correctly sorts a set of $n >= 1$ integers. The result remains in $a[0], \ldots, a[n-1]$ such that $a[0] <= a[1] <= \ldots <= a[n-1]$.
- Proof:

For $i = q$, following the execution of line 6-11, we have $a[q] <= a[r]$, $q < r < = n-1$.

For $i > q$, observing, $a[0], \ldots, a[q]$ are unchanged.

Hence, increasing i, for $i = n-2$, we have

$a[0] <= a[1] <= \ldots <= a[n-1]$

Recursive Algorithms

- Direct recursion
  - Functions call themselves
- Indirect recursion
  - Functions call other functions that invoke the calling function again
- When is recursion an appropriate mechanism?
  - The problem itself is defined recursively
  - Statements: if-else and while can be written recursively
  - Art of programming
- Why recursive algorithms ?
  - Powerful, express an complex process very clearly

Recursive Implementation of Binary Search

```
int binsearch(int list[], int searchnum, int left, int right)
{// search list[0]<= list[1]<=...<=list[n-1] for searchnum
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; break;
   } }
 return -1;}
```

```
int compare(int x, int y)
{
  if (x< y) return -1;
  else if (x== y) return 0;
  else return 1;
}
```

Recursive Implementation of Binary Search

```
int binsearch(int list[], int searchnum, int left, int right)
{// search list[0]<= list[1]<=...<=list[n-1] for searchnum
int middle;
 while (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;
}
```

Chapter 1 Basic Concepts

- Overview: System Life Cycle
- Algorithm Specification
- Data Abstraction
- Performance Analysis
- Performance Measurement

Data Abstraction

- Types of data
  - All programming language provide at least minimal set of predefined data type, plus user defined types
- Data types of C
  - Char, int, float, and double
    - may be modified by short, long, and unsigned
  - Array, struct, and pointer

Data Type

- Definition
  - A ***data type*** is a collection of ***objects*** and a set of ***operations*** that act on those objects
- Example of "int"
  - Objects: 0, +1, -1, ..., Int_Max, Int_Min
  - Operations: *arithmetic*(+, -, *, /, and %), *testing*(equality/inequality), *assigns*, *functions*
- Define operations
  - Its ***name***, possible ***arguments*** and ***results*** must be specified
- The design strategy for representation of objects
  - *Transparent* to the user

Abstract Data Type

- 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 operation.#
- Why abstract data type ?
  - implementation-independent

Classifying the Functions of a Data Type

- **Creator/constructor:**
  - Create a new instance of the designated type
- **Transformers**
  - Also create an instance of the designated type by using one or more other instances
- **Observers/reporters**
  - Provide information about an instance of the type, but they do not change the instance
- Notes
  - An ADT definition will include at least one function from each of these three categories

An Example of the ADT

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 is Nat_Number, TRUE, FALSE is Boolean and where .
+, -, <, and == are the usual integer operations

```
Nat_NoZero()             ::= 0
Boolean Is_Zero(x)       ::= if (x) return FALSE
Nat_No 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_No Successor(x)      ::= if (x== INT_MAX) return x
                             else return x+ 1
Nat_No Subtract(x, y)    ::= if (x< y) return 0
                             else return x-y
```

end Natural_Number

Chapter 1 Basic Concepts

- Overview: System Life Cycle
- Algorithm Specification
- Data Abstraction
- Performance Analysis
- Performance Measurement

Performance Analysis

- Performance evaluation
  - Performance **analysis**
  - Performance **measurement**
- Performance **analysis - prior**
  - an important branch of CS, ***complexity theory***
  - estimate ***time*** and ***space***
  - machine independent
- Performance **measurement - posterior**
  - The actual ***time*** and ***space*** requirements
  - machine dependent

Performance Analysis(Cont.)

- Space and time
  - Does the program efficiently use primary and secondary storage?
  - Is the program's running time acceptable for the task?
- Evaluate a program generally
  - Does the program ***meet*** the original ***specifications*** of the task?
  - Does it ***work correctly***?
  - Does the program contain ***documentation*** that show ***how to use it*** and ***how it works***?
  - Does the program ***effectively use functions*** to create logical units?
  - Is the program's code ***readable***?

Performance Analysis(Cont.)

- Evaluate a program
  - ***MWGWRERE***

    Meet specifications, **W**ork correctly,

    Good user-interface, **W**ell-documentation,

    Readable, **E**ffectively use functions,

    Running time acceptable, **E**fficiently use space
- How to achieve them?
  - Good programming style, experience, and practice
  - Discuss and think

Space Complexity

- Definition
  - The ***space complexity*** of a program is the amount of memory that it needs to run to completion
- The space needed is the sum of
  - ***Fixed*** space and ***Variable*** space
- Fixed space
  - Includes the instructions, variables, and constants
  - Independent of the number and size of I/O
- Variable space
  - Includes dynamic allocation, functions' recursion
- Total space of any program
  - $S(P) = c + \boldsymbol{S_p(Instance)}$

Examples of Evaluating Space

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

$S_{abc}(I) = 0$

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

$S_{sum}(I) = S_{sum}(n) = 0$

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

$S_{rsum}(n) = 4*n$

parameter:float(list[])	1
parameter:integer(n)	1
return address	1
return value	1

Time Complexity

- Definition
  - The ***time complexity, T(p),*** taken by a program P is the sum of the compile time and the run time
- Total time
  - T(P)= compile time + ***run*** (or ***execution***) ***time***

    $= c + t_p(\textit{instance characteristics})$

    Compile time does not depend on the instance characteristics
- How to evaluate?
  - Use the system clock
  - Number of ***steps*** performed
    - machine-independent
- Definition of a program step
  - A ***program step*** is a syntactically or semantically meaningful program segment whose execution time is independent of the ***instance*** characteristics

  (10 additions can be one step, 100 multiplications can also be one step)

  (p33~p35 有計算C++ 語法之 steps 之概述, 原則是一個表示式一步)

Examples of Determining Steps

- *the first method: count by a program*

```
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;
}
```

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

2n+ 3

Examples of Determining Steps(Cont.)

```
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++; //return
  return list[0];
}
```

$$t_{rsum}(0) = 2$$

$$\begin{aligned} t_{rsum}(n) &= 2 + t_{rsum}(n-1) \\ &= 2 + 2 + t_{rsum}(n-2) \\ &= 2*2 + t_{rsum}(n-2) \\ &= \ldots \\ &= 2n + t_{rsum}(0) = 2n+2 \end{aligned}$$

2n+ 2

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

p.39, program 1.19
自行計算

2rows*cols+ 2rows+ 1

Examples of Determining Steps(Cont.)

- The second method: build a table to count
  s/e: steps per execution
  frequency: total numbers of times each statements is executed

Statement	*s/e*	*Frequency*	*Total Steps*
void add(int a[][MaxSize], . . .	*0*	*0*	*0*
{	*0*	*0*	*0*
int i, j;	*0*	*0*	*0*
for (i=0; i< rows; 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***

Remarks of Time Complexity

- Difficulty: the time complexity is not dependent solely on the number of inputs or outputs
- To determine the step count
  - **Best case, Worst case,** and **Average**
- Example

```
int binsearch(int list[], int searchnum, int left, int right)
{// search list[0]<= list[1]<=...<=list[n-1] for searchnum
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;}
```

Asymptotic Notation(O, Ω, Θ)

- motivation
  - Target: Compare the time complexity of two programs that computing the same function and predict the growth in run time as instance characteristics change
  - Determining the exact step count is difficult task
  - Not very useful for comparative purpose

    ex: $C_1n^2+C_2n <= C_3n$ for $n <= 98$, ($C_1=1$, $C_2=2$, $C_3=100$)

    $C_1n^2+C_2n > C_3n$ for $n > 98$,
  - Determining the exact step count usually not worth(can not get exact run time)
- Asymptotic notation
  - Big "oh" O
    - upper bound(current trend)
  - Omega Ω
    - lower bound
  - Theta Θ
    - upper and lower bound

Asymptotic Notation O

- Definition of Big "oh"
  - $f(n) = O(g((n))$ iff there exist **positive** constants c and n_0 such that $f(n) <= cg(n)$ for all n, $n >= n_0$
- Examples
  - $3n+2 = O(n)$ as $3n+2 <= 4n$ for all $n >= 2$
  - $10n^2+4n+2 = O(n^2)$ as $10n^2+4n+2 <= 11n^2$ for $n >= 5$
  - $3n+2 <> O(1)$, $10n^2+4n+2 <> O(n)$
- Remarks
  - g(n) is the least upper bound
    - $n = O(n^2) = O(n^{2.5}) = O(n^3) = O(2^n)$
  - $O(1)$: constant, $O(n)$: linear, $O(n^2)$: quadratic, $O(n^3)$: cubic, and $O(2^n)$: exponential

Asymptotic Notation O (Cont.)

- Remarks on "="
  - $O(g(n)) = f(n)$ is meaningless
  - "=" as "*is*" and not as "equals"
- Theorem
  - If $f(n) = a_m n^m + \ldots + a_1 n + a_0$, then $f(n) = O(n^m)$
  - Proof:

[redacted]

Asymptotic Notation Ω

- Definition
  - $f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that $f(n) >= cg(n)$ for all n, $n >= n_0$
- Examples
  - $3n+2 = \Omega(n)$ as $3n+2 >= 3n$ for $n >= 1$
  - $10n^2+4n+2 = \Omega(n^2)$ as $10n^2+4n+2 >= n^2$ for $n >= 1$
  - $6*2^n+n^2 = \Omega(2^n)$ as $6*2^n+n^2 >= 2^n$ for $n >= 1$
- Remarks
  - The largest lower bound
    - $3n+3 = \Omega(1)$, $10n^2+4n+2 = \Omega(n)$; $6*2^n+n^2 = \Omega(n^{100})$
- Theorem
  - If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$

Asymptotic Notation Θ

- Definition
  - $f(n) = \Theta(g(n))$ iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) <= f(n) <= c_2 g(n)$ for all n, $n >= n_0$
- Examples
  - $3n+2=\Theta(n)$ as $3n+2>=3n$ for $n>1$ and $3n+2<=4n$ for all $n>= 2$
  - $10n^2+ 4n+ 2= \Theta(n^2)$; $6*2^n+n^2= \Theta(2^n)$
- Remarks
  - Both an upper and lower bound
  - $3n+2<>\Theta(1)$; $10n^2+4n+ 2<> \Theta(n)$
- Theorem
  - *If* $f(n)= a_m n^m+ \ldots +a_1 n+ a_0$ *and* $a_m> 0$, *then* $f(n)= \Theta(n^m)$

Example of Time Complexity Analysis

Statement	Asymptotic complexity
void add(int a[][Max.......)	0
{	0
int i, j;	0
for(i= 0; i< rows; i++)	Θ(rows)
for(j=0; j< cols; j++)	Θ(rows*cols)
c[i][j]= a[i][j]+ b[i][j];	Θ(rows*cols)
}	0
Total	Θ(rows*cols)

Example of Time Complexity Analysis(Cont.)

- The more global approach to count steps:
  focus the variation of instance characterics.

```
int binsearch(int list[], int .....)
{  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;
}
```

worst case $\Theta(\log n)$

Example of Time Complexity Analysis(Cont.)

```
void perm(char *a, int k, int n)
{//generate all the 排列 of
// a[k],…a[n-1]
char temp;
 if (k == n-1){
    for(int i= 0; i<=n; i++)
              cout << a[i]<<" ";
    cout << endl;
  }
else {
 for(i= k; i< n; i++){
  temp=a[k];a[k]=a[i];a[i]=temp;
  perm(a, k+1, n);
  temp=a[k];a[k]=a[i];a[i]=temp;
    }
  }
}
```

k= n-1, $\Theta(n)$

k< n-1, else

for loop, n-k times

each call $T_{perm}(k+1, n-1)$

hence, $\Theta(T_{perm}(k+1, n-1))$

so, $T_{perm}(k, n-1)= \Theta((n-k)(T_{perm}(k+1, n-1)))$

Using the substitution, we have

$T_{perm}(0, n-1)= \Theta(n(n!))$, n>= 1

Example of Time Complexity Analysis(Cont.)

- Magic square
  - An n-by-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
  - Example, n= 5(n must be odd)

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Magic Square (Cont.)

- Coxeter has given the simple rule
  - Put a one in the middle box of the top row.

    Go up and left assigning numbers in increasing order to empty boxes.

    If your move causes you to jump off the square, figure out where you would be if you landed on a box on the opposite side of the square.

    Continue with this box.

    If a box is occupied, go down instead of up and continue.

Magic Square (Cont.)

```
procedure MAGIC(square, n)
// for n odd create a magic square which is declared as an array
// square(0: n-1, 0: n-1)
// (i, j) is a square position. 2<= key <= n^2 is integer valued
if n is even the [print("input error"); stop]
SQUARE<- 0
square(0, (n-1)/2)<- 1;  // store 1 in middle of first row
key<- 2; i<- 0; j<- (n-1)/2  // i, j are current position
while key <= n^2 do
  (k, l)<- ((i-1) mod n, (j-1)mod n)  // look up and left
  if square(k, l) <> 0
     then i<- (i+1) mod n  // square occupied, move down
  else (i, j)<- (k, l)  // square (k, l) needs to be assigned
 square(i, j)<- key    // assign it a value
 key<- key + 1
end
print(n, square) // out result
end MAGIC
```

Practical Complexities

- Time complexity
  - Generally some function of the instance characteristics
- Remarks on "n"
  - If $Tp=\Theta(n)$, $Tq=\Theta(n^2)$, then we say P is faster than Q for "sufficiently large" n.
    - since $Tp<=cn$, $n>=n_1$, and $Tq<=dn_2$, $n>=n_2$, but $cn<=dn^2$ for $n>=c/d$

      so P is faster than Q whenever $n>=\max\{n_1, n_2, d/c\}$
  - See Table 1.7 and Figure 1.3
- For reasonable large n, n> 100, only program of small complexity, n, nlog n, n^2, n^3 are feasible
  - See Table 1.8

Table 1.8 Times on a 1 bsps computer

Time for f(n) instructions on 10^9 instr/sec computer

n	$f(n)= n$	$f(n)=\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01us	.03us	.1us	1us	10us	10s	1us
20	.02us	.09us	.4us	8us	160us	2.84hr	1ms
30	.03us	.15us	.9us	27us	810us	6.83d	1s
40	.04us	.21us	1.6us	64us	2.56ms	12136d	18.3m
50	.05us	.28us	2.5us	125us	6.25us	3.1y	13d
100	.10us	.66us	10us	1ms	100ms	3171y	$4*10^{13}$y
1,000	1.00us	0.96us	1ms	1s	16.67m	$3*10^{13}$y	$32*10^{283}$y
10,000	10.00us	130.03us	100ms	16.67m	115.7d	$3*10^{23}$y	
100,000	100.00us	1.66ms	10s	11.57d	3171y	$3*10^{33}$y	
1,000,000	1.00ms	19.92ms	16.67m	31.71y	$3*10^7$y	$3*10^{43}$y	

Table 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
nlog n	Log Linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	61	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
n!	Factorial	1	2	54	40326	20922789888000	$26313*10^{33}$

Chapter 1 Basic Concepts

- Overview: System Life Cycle
- Algorithm Specification
- Data Abstraction
- Performance Analysis
- Performance Measurement

Performance Measurement

- ❑Obtaining the actual space and time of a program
- ❑Using Borland C++, ‘386 at 25 MHz
- ❑Time(hsec): returns the current time in hundredths of a sec.
- ❑Goal: 得到測量結果的曲線圖, 並進而求得執行時間方程式

  Step 1, 分析$\Theta(g(n))$, 做為起始預測

  Step 2, write a program to test

  -技巧1 : to time a short event, to repeat it several times

  -技巧2 : suitable test data need to be generated

Example:
```
time(start);
for(b=1; b<=r[j];b++)
    k=seqsearch(a,n[j],0);// 被測對象
time(stop);
totaltime = stop –start;
runtime = totaltime/r[j]; // 結果參考fig 1.5, fig1.6
```

Summary

- Overview: System Life Cycle
- Algorithm Specification
  - Definition, Description
- Data Abstraction - ADT
- Performance Analysis
  - Time and Space
    - O(g(n))
- Performance Measurement
- Generating Test Data

  \- analyze the algorithm being tested to determine classes of data