DATA STRUCTURES AND ALGORITHMS

Textbook:
Fundamentals of Data Structure in C++, Silicon Press, 2006

Instructor:
吕建华, CSE & COSE
Email: lujianhua@seu.edu.cn
Tele.: 13585112527
Room 216, Computer Building
Teaching assistants:

王润（二班） 827632740@qq.com
陈坤（四班） 1104379060@qq.com
Total class hours:  56
week 1-16

Total lab. hours:   16

All on Wednesday evening 6:30pm-9:30pm
of week 5,  8, 11, 14.

At Room 262,  266, Computer Building
Assignments and projects:

• Should be handed to teaching assistants.
Evaluation:
Course Attendance: 10%,
Exercises and Projects: 20%
Final Examination (Textbook and Course Notes allowed): 70%
References:

1 金远平, 数据结构( C++描述), 清华大学出版社, 2005

Prerequisites:

1 Programming Language: C, C++
Tips

- Make good use of your time in class
  - Listening
  - Thinking
  - Taking notes
- Expend your free time
  - Go over
  - Programing
- Take a pen and some paper with you
  - Notes
  - Exercises
物有本末，事有终始。知所先后，则近道矣。
In Computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.
Sorting

- Rearrange $a[0], a[1], \ldots, a[n-1]$ into ascending order. When done, $a[0] \leq a[1] \leq \ldots \leq a[n-1]$
- $8, 6, 9, 4, 3 \Rightarrow 3, 4, 6, 8, 9$
Sort Methods

- Insertion Sort
- Bubble Sort
- Selection Sort
- Counting Sort
- Shell Sort
- Heap Sort
- Merge Sort
- Quick Sort
- ......
Insert An Element

- Given a sorted list/sequence, insert a new element
  - Given 3, 6, 9, 14
  - Insert 5
  - Result 3, 5, 6, 9, 14
Insert an Element

- 3, 6, 9, 14 insert 5
- Compare new element (5) and last one (14)
- Shift 14 right to get 3, 6, 9, , 14
- Shift 9 right to get 3, 6, , 9, 14
- Shift 6 right to get 3, , 6, 9, 14
- Insert 5 to get 3, 5, 6, 9, 14
// insert t into a[0:i-1]
int j;
for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];
a[j + 1] = t;
Insertion Sort

- Start with a sequence of size 1
- Repeatedly insert remaining elements
Insertion Sort

- Sort 7, 3, 5, 6, 1
- Start with 7 and insert 3 => 3, 7
- Insert 5 => 3, 5, 7
- Insert 6 => 3, 5, 6, 7
- Insert 1 => 1, 3, 5, 6, 7
Insertion Sort

for (int i = 1; i < a.length; i++)
{
    // insert a[i] into a[0:i-1]
    // code to insert comes here
}
Insertion Sort

for (int i = 1; i < a.length; i++)
{// insert a[i] into a[0:i-1]
    int t = a[i];
    int j;
    for (j = i - 1; j >= 0 && t < a[j]; j--)
        a[j + 1] = a[j];
    a[j + 1] = t;
}
for (int i = 1; i < a.length; i++)
{ // insert a[i] into a[0:i-1]
    int t = a[i];
    int j;
    for (j = i - 1; j >= 0 && t < a[j]; j--)
        a[j + 1] = a[j];
    a[j + 1] = t;
}
Purpose:
Provide the tools and techniques necessary to design and implement large-scale software systems, including:

- Data abstraction and encapsulation
- Algorithm specification and design
- Performance analysis and measurement
- Recursive programming
Overview: System Life Cycle

(1) Requirements
    specifications of purpose
    input
    output

(2) Analysis
    break the problem into manageable pieces
    bottom-up
    top-down
Overview: System Life Cycle

(3) Design
   a SYSTEM? (from the designer’s angle)
   data objects
   operations on them

TO DO
   abstract data type
   algorithm specification and design

Example: scheduling system of university

??
??
(4) Refinement and coding representations for data object algorithms for operations components reuse

(5) Verification and maintenance correctness proofs testing error removal update
Data Abstraction and Encapsulation

Data Encapsulation or information Hiding is the concealing of the implementation details of a data object from the outside world.

Data Abstraction is the separation between the specification of a data object and its implementation.

DVD example.
A Data Type is a collection of objects and a set of operations that act on those objects.

predefined and user-defined:
char, int, arrays, structs, classes.

An Abstract Data Type (ADT) is a data type with the specification of the objects and the specification of the operations on the objects being separated from the representation of the objects and the implementation of the operations.
Benefits of data abstraction and data encapsulation:

(1) Simplification of software development
   Application: data types A, B, C & Code Glue
   (a) a team of 4 programmers
   (b) a single programmer
Testing and debugging

Code with data abstraction

Code without data abstraction

Unshaded areas represent code to be searched for bugs.
(3) Reusability

data structures implemented as distinct entities of a software system

(4) Modifications to the representation of a data type

a change in the internal implementation of a data type will not affect the rest of the program as long as its interface does not change.
An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.
Must satisfy the following criteria:

(1) **Input**  Zero or more quantities externally supplied.

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

(3) **Definiteness**  Clear and unambiguous.

(4) **Finiteness**  Terminates after a finite number of steps.

(5) **Effectiveness**  Basic enough, feasible

Compare: algorithms and programs

**Finiteness**
Recursion
Exercises: P32-2, P33-14
Performance Analysis and Measurement

Definition:
The Space complexity of a program is the amount of memory it needs to run to completion.
The Time complexity of a program is the amount of computer time it needs to run to completion.

(1) Priori estimates --- Performance analysis
(2) Posteriori testing--- Performance measurement
Performance Analysis

Space complexity

The space requirement of program $P$:

$S(P)=c+S_P$ (instance characteristics)

We concentrate solely on $S_P$. 
Performance Analysis

Example 1.10

float Rsum (float *a, const int n) //compute $\sum_{i=0}^{n-1} a[i]$ recursively
{
    if (n <=0) return 0;
    else return (Rsum(a,n-1)+a[n-1]);
}

$\sum_{i=0}^{n-1} a[i] = n$
The instances are characterized by

\[ n \]

each call requires 4 words (n, a, return value, return address)

the depth of recursion is

\[ n+1 \]

\[ S_{rs\text{sum}}(n) = 4(n+1) \]
Time complexity

Run time of a program $P$:

$$T(P) = c + t_p \text{(instance characteristics)}$$

A **program step** is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is **independent** of instance characteristics.

In P41-43 of the textbook, there is an detailed assignment of step counts to statements in C++.
Step Count

A step is an amount of computing that does not depend on the instance characteristic $n$.

10 adds, 100 subtracts, 1000 multiplies can all be counted as a single step.

$n$ adds cannot be counted as 1 step.
Our main concern:

how many steps are needed by a program to solve a particular problem instance?

2 ways:

(1) count

(2) table
Example 1.12

```c
float Rsum (float *a, const int n) {
    count++; // for if
    if (n <= 0) {
        count++; // for return
        return 0;
    }
    else {
        count++; // for return
        return (Rsum(a,n-1)+a[n-1]);
    }
}

\[ t_{Rsum}(0) = 2, \]
\[ t_{Rsum}(n) = 2 + t_{Rsum}(n-1) \]
\[ = 2 + 2 + t_{Rsum}(n-2) \]
\[ = 2n + t_{Rsum}(0) \]
\[ = 2n + 2 \]
Example 1.14: Fibonacci numbers

```cpp
void Fibonacci(int n) {
    // compute the Fibonacci number F
    if (n <= 1) {
        cout << n << endl;
    } else {
        int fn = 0, fnm2 = 0, fnm1 = 1;
        for (int i = 2; i <= n; i++) {
            fn = fnm1 + fnm2;
            fnm2 = fnm1;
            fnm1 = fn;
        }
        cout << fn << endl;
    }
}
```

Let us use a table to count its total steps.

<table>
<thead>
<tr>
<th>Line</th>
<th>s/e</th>
<th>frequency</th>
<th>total steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>n-1</td>
<td>n-1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>n-l</td>
<td>n-l</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>n-l</td>
<td>n-l</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>n-l</td>
<td>n-l</td>
</tr>
</tbody>
</table>

Note: The table assumes that the Fibonacci function is called with a specific input.

The table shows the number of times each line is executed and its corresponding total steps.

In the Fibonacci function, the if statement is executed if the input `n` is less than or equal to 1, and the else block is executed otherwise.

The table helps to visualize the total steps taken by the function, which is the sum of the number of times each line is executed and its corresponding frequency.
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>n-1</td>
<td>n-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>n-1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So

for \( n > 1 \), \( t_{\text{Fibonacci}}(n) = 4n + 1 \),
for \( n = 0 \) or 1, \( t_{\text{Fibonacci}}(n) = 2 \)
Sometime, the instance characteristics is related with the content of the input data set.

e.g., *BinarySearch*.

Hence:

- best-case
- worst-case,
- average-case.
Asymptotic Notation

Because of the inexactness of what a step stands for, we are mainly concerned with the magnitude of the number of steps.

**Definition [O]:** $f(n) = O(g(n))$ iff there exist positive constants $c$ and $n_0$ such that $f(n) \leq c \cdot g(n)$ for all $n$, $n > n_0$.

**Example 1.13:** $3n + 2 = O(n)$, $6 \cdot 2^n + n^2 = O(2^n)$, …
Note $g(n)$ is an upper bound.

$n=O(n^2)$, $n=O(2^n)$, …,

for $f(n)=O(g(n))$ to be informative, $g(n)$ should be as small as possible.

In practice, the coefficient of $g(n)$ should be 1. We never say $O(3n)$. 
Theory 1.2: if \( f(n) = a_m n^m + \ldots + a_1 n + a_0 \), then \( f(n) = O(n^m) \).

When the complexity of an algorithm is actually, say, \( O(\log n) \), but we can only show that it is \( O(n) \) due to the limitation of our knowledge, it is OK to say so. This is one benefit of \( O \) notation as upper bound.

Self-study:

\( \Omega \) --- low bound

\( \Theta \) --- equal bound
A Few Comparisons

Function #1

\[ n^3 + 2n^2 \]
\[ n^{0.1} \]
\[ n + 100n^{0.1} \]
\[ 5n^5 \]
\[ n^{-15}2^n/100 \]
\[ 8^{2\log n} \]

Function #2

\[ 100n^2 + 1000 \]
\[ \log n \]
\[ 2n + 10 \log n \]
\[ n! \]
\[ 1000n^{15} \]
\[ 3n^7 + 7n \]
Race I

\( n^3 + 2n^2 \) \hspace{1cm} \text{vs.} \hspace{1cm} 100n^2 + 1000
Race II

$n^{0.1}$ vs. $\log n$
Race III

\[ n + 100n^{0.1} \quad \text{vs.} \quad 2n + 10 \log n \]
Race IV

$5n^5$ vs. $n!$
Race V

$n^{-15}2^n/100$ vs. $1000n^{15}$
Race VI

$8^{2 \log(n)}$ vs. $3n^7 + 7n$
The Losers Win

Function #1

\[ n^3 + 2n^2 \]
\[ n^{0.1} \]
\[ n + 100n^{0.1} \]
\[ 5n^5 \]
\[ n^{-15}2^n/100 \]
\[ 8^{2\log n} \]

Better algorithm!

Function #2

\[ 100n^2 + 1000 \]
\[ \log n \]
\[ 2n + 10 \log n \]
\[ n! \]
\[ 1000n^{15} \]
\[ 3n^7 + 7n \]

\[ O(n^2) \]
\[ O(\log n) \]
\[ \text{TIE} \]
\[ O(n) \]
\[ O(n^5) \]
\[ O(n^{15}) \]
\[ O(n^6) \]
Common Names

constant: \( O(1) \)
logarithmic: \( O(\log n) \)
linear: \( O(n) \)
log-linear: \( O(n \log n) \)
quadratic: \( O(n^2) \)
polynomial: \( O(n^k) \) \( (k \text{ is a constant}) \)
exponential: \( O(c^n) \) \( (c \text{ is a constant } > 1) \)
Practical Complexity

How the various functions grow with n?

Ultimate Laptop, 1 year
1 second
1000 MIPS, since Big Bang
1000 MIPS, 1 day
<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>$f(n)=n^4$</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>10 µs</td>
<td>10s</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>160 µs</td>
<td>2.84h</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>810 µs</td>
<td>6.83d</td>
<td>1 s</td>
</tr>
<tr>
<td>40</td>
<td>0.04 µs</td>
<td>0.21 µs</td>
<td>1.6 µs</td>
<td>2.56ms</td>
<td>121d</td>
<td>18m</td>
</tr>
<tr>
<td>50</td>
<td>0.05 µs</td>
<td>0.28 µs</td>
<td>2.5 µs</td>
<td>6.25ms</td>
<td>3.1y</td>
<td>13 d</td>
</tr>
<tr>
<td>100</td>
<td>0.1 µs</td>
<td>0.66 µs</td>
<td>10 µs</td>
<td>100 ms</td>
<td>3171y</td>
<td>4*10^{13}y</td>
</tr>
<tr>
<td>$10^3$</td>
<td>1 µs</td>
<td>9.66 µs</td>
<td>1ms</td>
<td>16.67m</td>
<td>4*10^{13}y</td>
<td></td>
</tr>
<tr>
<td>$10^4$</td>
<td>10 µs</td>
<td>130 µs</td>
<td>100ms</td>
<td>115.7d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^5$</td>
<td>100 µs</td>
<td>1.66ms</td>
<td>10s</td>
<td>3171y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 1.8: Times on a 1-billion-steps-per-second computer**
Performance Measurement

Performance measurement is concerned with obtaining the actual space and time requirements of a program.

To time a short event it is necessary to repeat it several times and divide the total time for the event by the number of repetitions.
Let us look at the following program:

```c
int SequentialSearch (int *a, const int n, const int x )
{
    // Search a[0:n-1].
    int i;

    for (i=0; i < n && a[i] != x; i++)
    {
        if (i == n) return -1;
        else return i;
    }
}
```
void TimeSearch() {
    int a[1000], n[20];
    const long r[20] = {300000, 300000, 200000, 200000, 100000, 100000, 100000, 80000, 80000, 50000, 50000, 
                        25000, 15000, 15000, 10000, 7500, 7000, 6000, 5000, 5000};

    for (int j = 0; j < 1000; j++) a[j] = j + 1; // initialize a
    for (j = 0; j < 10; j++) { // values of n
        n[j] = 10 * j; n[j + 10] = 100 * (j + 1);
    }

    cout << "n total runTime" << endl;
}
for (j=0; j<20; j++) {
    long start, stop;
    time (&start); // start timer
    for (long b=1; b<=r[j]; b++)
        int k = seqsearch(a, n[j], 0); // unsuccessful search
    time (&stop); // stop timer
    long totalTime = stop - start;
    float runTime = (float) (totalTime) / (float) (r[j]);
    cout << " " << n[j] << " " << totalTime << " " << runTime
         << endl;
}

The results of running TimeSearch are as in the next slide.
<table>
<thead>
<tr>
<th>n</th>
<th>total</th>
<th>runTime</th>
<th>n</th>
<th>total</th>
<th>runTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>241</td>
<td>0.0008</td>
<td>100</td>
<td>527</td>
<td>0.0105</td>
</tr>
<tr>
<td>10</td>
<td>533</td>
<td>0.0018</td>
<td>200</td>
<td>505</td>
<td>0.0202</td>
</tr>
<tr>
<td>20</td>
<td>582</td>
<td>0.0029</td>
<td>300</td>
<td>451</td>
<td>0.0301</td>
</tr>
<tr>
<td>30</td>
<td>736</td>
<td>0.0037</td>
<td>400</td>
<td>593</td>
<td>0.0395</td>
</tr>
<tr>
<td>40</td>
<td>467</td>
<td>0.0047</td>
<td>500</td>
<td>494</td>
<td>0.0494</td>
</tr>
<tr>
<td>50</td>
<td>565</td>
<td>0.0056</td>
<td>600</td>
<td>439</td>
<td>0.0585</td>
</tr>
<tr>
<td>60</td>
<td>659</td>
<td>0.0066</td>
<td>700</td>
<td>484</td>
<td>0.0691</td>
</tr>
<tr>
<td>70</td>
<td>604</td>
<td>0.0075</td>
<td>800</td>
<td>467</td>
<td>0.0778</td>
</tr>
<tr>
<td>80</td>
<td>681</td>
<td>0.0085</td>
<td>900</td>
<td>434</td>
<td>0.0868</td>
</tr>
<tr>
<td>90</td>
<td>472</td>
<td>0.0094</td>
<td>1000</td>
<td>484</td>
<td>0.0968</td>
</tr>
</tbody>
</table>

Times in hundredths of a second, the plot of the data can be found in Fig. 1.7.
Issues to be addressed:
(1) Accuracy of the clock
(2) Repetition factor
(3) Suitable test data for worst-case or average performance
(4) Purpose: comparing or predicting?
(5) Fit a curve through points

Exercises:
P72-10