Data Structures and Algorithms

Lecturer:	Mr. Van Ting	Dr. Derek Pao
Office:	G6418	G6514
Email:	cwting@cityu.edu.hk	d.pao@cityu.edu.hk
Tel:	3442-7246	3442-8607

Tentative syllabus

- Review of basic concepts
  - Scalar, Pointer, and Reference
  - Parameter passing in function call
  - Arrays
  - Loop design
  - Relationship between data structure and algorithm
  - Computation complexity
  - Class
  - C vs. C++, and Java vs. C++
- Linked lists
- Stacks and Queues
- C++ Standard Template Library
- Recursion
- Binary trees
- Heap and priority queue
- Sorting and Searching algorithms
- Hash table

Intended Learning Outcomes

Students should be able to

- apply structural programming approach to solve more complex computation problems
- demonstrate applications of standard data structures such as linked list, stack, queue and tree
- solve computation problems using recursion where appropriate
- understand different sorting and searching algorithms

References:

C++ on-line tutorial, http://www.cplusplus.com/doc/tutorial/

D. S. Malik, Data Structures Using C++, 2nd Edition, 2010.

Assessment

Examination: 60%

Coursework: 40%

- Programming Assignments – 20%
- Test – 15%
- Tutorial Attendance & Performance (5%)

In order to pass the course, one should obtain

(i) at least 35% in examination,
(ii) at least 35% in coursework,
(iii) 40% in overall marks.

Copying of assignments is strictly prohibited.
If the submitted program of student *A* is found to be the "same" as that of student *B*, both students *A* and *B* will receive zero mark for that assignment. The scores for Tutorial Attendance & Performance will also be reduced to zero as an additional penalty.

General opinions of students on EE2331:

Student: At the beginning I am comfortable with the materials. But I skip a class in week 3, and then I don't quite understand the discussion for the rest of the semester.

Teacher: Materials discussed in this course are highly correlated. So, don't skip any classes.

Student: I have good grade in CS2363. But I find EE2331 a lot more difficult.

Teacher: EE2331 is a very basic course in computing. It is not difficult. But you have to adjust your learning method.

We emphasize on problem solving rather than syntax.

A computer program cannot be constructed by copy-and-paste. You have to understand the methods, and be able to apply what you have learnt in a different problem setting. It is counter-productive in trying to memorize the program codes.

Student: I understand the materials presented by the instructor in lectures. But I just can't write my own programs.

Teacher: There are different levels of understanding. You have to be able to generalize the example solutions discussed in class to solve other problems.

You must acquire the following skills

- how to formulate a solution method to solve a problem
- how to organize and express your ideas in a systematic manner
- how to translate your ideas to program codes
- the ability to trace the execution of program codes by paper and pencil (without this skill, you can never find out why your program doesn't work)
- these skills cannot be learnt by just memorizing the notes!!

Student: My classmate(s) can finish the tutorials or assignments very quickly. But I have spent many hours or even days in front of the computer and cannot get my program working.

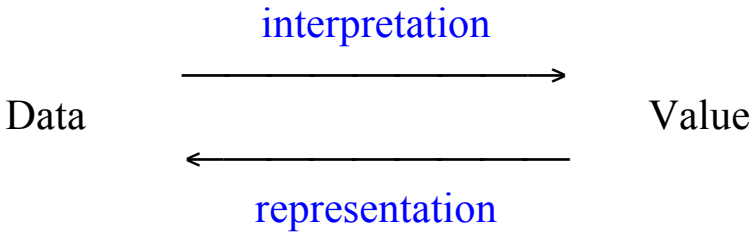
Teacher: Let's take "swimming" as an analogy. You can memorize the notes on "how to swim" in a couple of minutes. But you cannot learn how to swim by this way.

Programming skills can only be acquired via practices.
No pain, no gain.

You have to

- pay attention in class
- follow the guidelines given to you
- do the programming exercises yourself
- learn from your own mistakes
- do not take the short-cut

Data versus Value



Examples:

Data	Value	interpretation/representation
0100 0001	char 'A'	8-bit ASCII
	65	signed or unsigned integer
1111 1111	255	unsigned 8-bit integer
	–1	signed 8-bit integer (2's complement)
	–127	signed 8-bit integer (sign-magnitude)

Fundamental (Primitive) data types in C++ (for today's PC/compiler)

data type	size (byte)	interpretation/representation	range of values
bool	1	Boolean (not available in C)	false or true
char	1	signed number (2's complement)	–128 to 127
unsigned char		unsigned number	0 to 255
int	4	signed number (2's complement)	-2^{31} to $2^{31}-1$
unsigned int		unsigned number	0 to $2^{32}-1$
short	2	signed number (2's complement)	-2^{15} to $2^{15}-1$
unsigned short		unsigned number	0 to $2^{16}-1$
long	4	signed number (2's complement)	-2^{31} to $2^{31}-1$
unsigned long		unsigned number	0 to $2^{32}-1$
long long	8	signed number (2's complement)	-2^{63} to $2^{63}-1$
unsigned long long		unsigned number	0 to $2^{64}-1$
float	4	IEEE 32-bit floating point number	$\pm 1.4\times10^{-45}$ to $\pm 3.4\times10^{38}$
double	8	IEEE 64-bit floating point number	$\pm 5\times10^{-324}$ to $\pm 1.798\times10^{308}$
pointer	4	memory address	0 to $2^{32}-1$

Operators in C++ (not a complete list)

Operator	Symbol	Description
Assignment	=	
Arithmetic	+, -, *, /, %	
Increment, decrement	++, --	
Unary minus	-	
Comparison	==, !=, <, <=, >, >=	
Logical	!, &&, ||	
Bitwise	~, &, |, ^, <<, >>	
insertion, extraction	ostream << s istream >> i	insertion to an output stream, extraction from an input stream
Member and pointer	x[i]	subscript (x is an array or a pointer)
	*x	indirection, dereference (x is a pointer)
	&x	reference (address of x)
	x->y	Structure dereference (x is a pointer to object/struct, member y of object/struct pointed to by x)
	x.y	Structure reference (x is an object or struct, member y of x)

Scalar variables and pointers

Example 1

```
int a, b, c;
int *p;  /* p is an integer pointer
            p stores the address of an integer variable
            p points to an integer */

a = 1;  /* assign the value 1 to a    */
b = 2;  /* b is assigned the value 2 */
p = &c; /* &c refers to the address of c
           assign the address of c to p */

*p = a + b;  /* *p refers to the contents at location p
                Effect: value of c is set to 3 */
```

Example 2

```
int a, b, c;
int *p;  // initially p = NULL or undefined

a = 1;
b = 2;
*p = a + b;  /* Error: null-pointer exception.
                Cannot store a value to an undefined
                memory location */
```

Example 3

```
double d;
int *p;

p = &d;  /* Error: type mismatch.
            An integer pointer cannot be used to point
            to a double precision number.  */
```

Integer arithmetic (Be careful with the use of integer division)

```
int a = 3;
int b = 2;
int c = 4;

cout  << a / b * c << endl;  // output is 4 !!
cout  << a * c / b << endl;  // output is 6
```

Parameter passing in function call

1. Pass by value

```
void swap (int x, int y)
{    int t;

     t = x;
     x = y;
     y = t;
}

void main()
{    int i = 1;
     int j = 2;

     swap(i, j);
     // pass by value - copy the values of i and j to swap()
     // after the function return, i = 1, j = 2.
}
```

2. Pass by pointer in C++ (pass by reference in C)

In the C language, reference to a formal parameter is explicitly specified as a pointer.

```
void swap (int *x, int *y)  //x and y are integer pointers
{    int t;

     t = *x;
     *x = *y;
     *y = t;
}

void main()
{    int i = 1;
     int j = 2;

     swap(&i, &j);
     // &i represents the address (reference) of i
     // pass the addresses of i and j to swap()
     // after the function return, i = 2, j = 1.
}
```

3. Pass by reference in C++ (NOT supported in C)

In C++, you can specify a formal parameter in the function signature as a reference parameter.

```
void swap (int& x, int& y) //x and y are passed by reference
{    int t;

     t = x;
     x = y;
     y = t;
}

void main()
{    int i = 1;
     int j = 2;

     swap(i, j);
     // compiler finds that swap() uses pass by reference,
     // in the function call, the references of i
     // and j are passed to swap()
     // after the function return, i = 2, j = 1.
}
```

Reference datatype

```
int i = 2;

int& r = i;    //r is a reference to an integer
               //an initial value must be provided in the
               //declaration of r,
               //exception: member variable of a class

int *p = &i;  //p is a pointer to an integer

cout << "value of i = " << i << endl;
cout << "value of r = " << r << endl;
cout << "value of p = " << p << endl;
cout << "value of *p = " << *p << endl;

r = 4;
cout << "Execute r = 4" << endl;
cout << "value of i = " << i << endl;
cout << "value of r = " << r << endl;
```

```
Outputs produced:

value of i = 2
value of r = 2
value of p = 001AF9C0
value of *p = 2
Execute r = 4
value of i = 4
value of r = 4
```

C++ references differ from pointers in several essential ways:

- It is not possible to refer directly to a reference object after it is defined; any occurrence of its name refers directly to the object it references.
- Once a reference is created, it cannot be later made to reference another object; it cannot be *reseated*. This is often done with pointers.
- References cannot be *null*, whereas pointers can; every reference refers to some object, although it may or may not be valid. Note that for this reason, containers of references are not allowed.
- References cannot be uninitialized. Because it is impossible to reinitialize a reference, they must be initialized as soon as they are created. In particular, local and global variables must be initialized where they are defined, and references which are data members of class instances must be initialized in the initializer list of the class's constructor.

Arrays
All elements of an array have the same fixed, predetermined size.

For example:

```
int b[100]; //static allocation (compile-time)
int *a = new int[100]; //dynamic allocation (run-time)
```

reserve 100 successive memory locations, each large enough to store a single integer (typically an integer variable occupies 4 bytes in today's computer).

Array mapping function:
Address of the first element in an array is called the *base address* of the array.
Today's PCs are byte-addressable (address value refers to a specific byte).

Let *esize* be the size of the array element (in terms of bytes).
address of $b[i]$ = $base(b) + i * esize$

The size of int = 4 bytes.

In the C/C++ language an array variable in a function call is interpreted as a pointer variable. The following 2 implementations of the function `sum` are equivalent.

```
int sum_1(int a[], int n)
{
    int t = 0;
    for (int i = 0; i < n; i++)
        t += a[i];  //add the value stored at location
                    //base(a) + i*esize to variable t
    return t;
}


int sum_2(int *a, int n)
{
    int t = 0;
    for (int i = 0; i < n; i++)
        t += a[i];  //same as t += *(a+i)
        //a+i = physical address of a+i*esize
        //esize is implied by the data type
    return t;
}
```

Common errors made by students in array declaration:

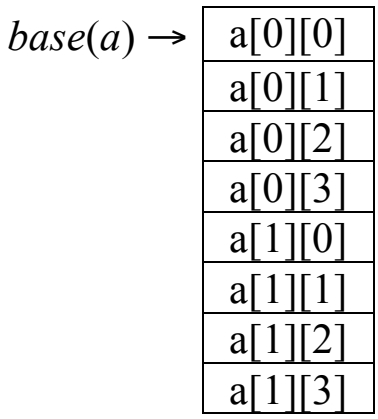
```
int n=100; // initialization of n is done during run-time
int A[n];  // Error: dynamic array size not allowed in
           // compile-time
```

2D Arrays

```
#define Rows 2
#define Cols 4
int a[Rows][Cols];  // an array with 2 rows & 4 columns
```

Multi-dimensional arrays are mapped to the linear address space of the computer system.

In C/C++, elements of a multi-dimensional array are arranged in row-major order.



Array mapping function:

base(*a*) = address of *a*[0][0]
address of *a*[i][j] = *base*(*a*) + (i × Cols + j) × *esize*

i = row index
j = column index
Cols = number of columns in a row
esize = size of an element (no. of bytes)

Array of pointers vs 2D-array

```
#define Rows 10
#define Cols 20
int a[Rows][Cols]; //a is a real 2D array with 200 elements
int *b[Rows];      //b is an array of 10 integer pointers

int i, j, k;

for (i = 0; i < Rows; i++) //create 10 linear arrays
   b[i] = new int[Cols];

k = 1;

//syntactically, you can access elements in b as b[i][j]
for (i = 0; i < Rows; i++)
   for (j = 0; j < Cols; j++)
   {  b[i][j] = k++;  //b[0][0] = 1;
                      //b[0][1] = 2; and so on

      a[i][j] = b[i][j];
   }
```

Remark: Although syntactically a[][] and b[][] appear to be the same, their physical representations are different.

Some programmers may instead define logical 2D array as

```
int **b;     //int *b[]
char **name; //char *name[]
```

This form is more convenient when you need to dynamically create (logical) 2D arrays of variable sizes, or array of variable-length character strings.

Example:

```
char *month[] = {"Illegal month", "January", "February",
                 "March", "April", "May", "June", "July",
                 "August", "September", "October",
                 "November", "December"};
```

Simple examples on loop design for the processing of arrays.

for-loop, while-loop, and do-loop

For-loop:

```
for (initialization; loop_test; loop_counting)
{
     //loop-body
}
```

Typical uses: compute the sum of an array with *n* elements

```
sum = 0;  //initialization
for (i = 0; i < n; i++)
   sum += A[i];
```

A for-loop can be replaced by a while-loop

```
initialization_statement;
while (loop_test)
{
     //loop-body

     loop_counting;
}
```

Using a while-loop to compute the sum of an array:

```
sum = 0;
i = 0;
while (i < n)
{
   sum += A[i];
   i++;
}
```

do-loop

```
sum = 0;
i = 0;
do {
    sum += A[i];
    i++;
} while (i < n);
```

The above do-loop may produce error if the length of the array is equal to zero, i.e. `n == 0;`

Use a do-loop only when you are 100% sure that the loop-body would be executed at least once for all possible input data.

DO NOT use `!= (Not equal)` to test the end of a range

```
for (i = 1; i != n; i++)
{
   //loop body
}
```

The test `i != n` is a poor idea.

How does the loop behave if `n` happens to be zero or negative?

The remedy is simple. Use `<=` rather than `!=` in the test condition:

```
for (i = 1; i <= n; i++)
{
   //loop body
}
```

Another bad programming style is to modify the value of the loop-counter inside the loop body of a for-loop.

```
for (i = 1; i <= n; i++)
{
   //main body of the loop

   if (testCondition)
      i = i + displacement;

   //i++ is executed before going back to top of the loop
}
```

Examples loop-design

Find the maximum value in an array of integers

Common mistake of students:

```
int findMax(int A[], int n)  //n = no. of elements in A[]
{
   int max; //variable to store the result

   max = 0;
   for (int i = 0; i < n; i++)
      if (A[i] > max)
         max = A[i];

   return max; //wrong result if all numbers in A[] are -ve
}
```

In the following examples, I would like to draw your attention to the uses of
preconditions – properties of the input data
postconditions – results to be produced by the loop, and
assertions – properties that are true at specific program locations.

Also, the examples serve to illustrate the relations between "data structures" and "algorithms".

Find the minimum value in an array of integers

Version 1: unordered array

```
//  precondition: n > 0 and A[] is unordered
int findMin_1(int A[], int n)  //n = no. of elements in A[]
{
   int min; //var to store the result

   min = A[0];
   for (int i = 1; i < n; i++)
      //assert: min = smallest value in A[0..i-1]
      if (A[i] < min)
         min = A[i];

   //postcondition: when the loop terminates,
   //min = smallest value in A[0..n-1]

   return min;
}
```

Version 2: ordered array

```
//  precondition: n > 0 and A[] is in ascending order
int findMin_2(int A[], int n)  //n = no. of elements in A[]
{
   //assert: A[0] is the smallest number in A[0..n-1]
   return A[0];
}
```

Find the location of the minimum value in an array of integers

```
//  precondition: n > 0 and A[] is unordered
int findMinLoc_1(int A[], int n)
{
   int min;

   min = 0;
   for (int i = 1; i < n; i++)
      // assert: A[min] = smallest value in A[0..i-1]
      if (A[i] < A[min])
         min = i;

   return min;
}
```

```
//  precondition: n > 0 and A[] is in ascending ordered
int findMinLoc_2(int A[], int n)
{
   return 0;
}
```

Remove duplicated values in an array of integers

Unordered array

```
A[] = {3, 6, 4, 7, 3, 3, 5, 6, 4, 3, 2, 8}
n = 12

After removing duplicated values
A[] = {3, 6, 4, 7, 8, 2, 5}
n = 7
```

Note that the order of the numbers in the resultant array is not important, because the array is unordered.

```
// Precondition: n > 0 and A[] is unordered
void removeDup(int A[], int& n) // n is passed by reference
{

   for (int i = 0; i < n-1; i++)
   {
      //assert: A[0..i] are distinct
      int j = i+1;
      while (j < n)  //should not use for-loop, why ??
      {
         if (A[j] == A[i])
         {
            //move the last element to location j
            A[j] = A[n-1];
            n -= 1;  //no. of element in A[] is reduced
         }
         else
            j++;
      }
   }
}
/* Postcondition: elements in A[0..n-1] are distinct, and
   A[] is unordered. */
```

Ordered array

```
A[] = {2, 3, 3, 3, 3, 4, 4, 5, 6, 6, 7, 8}
n = 12

After removing duplicated values
A[] = {2, 3, 4, 5, 6, 7, 8}
n = 7
```

Property of the array should be preserved.

Numbers in A[] should be maintained in ascending order.

```
// Precondition: n > 0 and A[] is in ascending ordered
void removeDup_2(int A[], int& n)
{
   int i = 0;
   while (i < n)
   {
      //assert: A[0..i] are distinct and ordered
      int j;
      for (j = i+1; j < n && A[j] == A[i]; j++)
         ;

      //assert: A[j] != A[i] and A[i..j-1] have equal value
      if (j > i+1) //(j-i) > 1 copies of the same number
      {
         //shift A[j..n-1] to the left by j-i-1 slots
         for (int k = j, int t = i+1; k < n; k++, t++)
            A[t] = A[k];

         n -= (j-i-1);  //j-i-1 elements are removed
      }
      i++;
   }
}
/* Postcondition: elements in A[] are distinct and A[] is
   in ascending order. */
```

Merge two sorted arrays of integer

```
//Precondition: A[] and B[] are sorted in ascending order

void merge(int A[], int na, int B[], int nb, int C[],
           int& nc)  // nc is passed by reference
{
   int i, j, k;

   i = j = k = 0;
   while (i < na && j < nb)
   {
      //assert: C[0..k-1] is sorted in ascending order
      if (A[i] <= B[j])
      {   C[k] = A[i];  // the 3 statements can be replaced
          k++;          // by C[k++] = A[i++];
          i++;
      }
      else
         C[k++] = B[j++];
   }

   //copy the remaining data in A[] or B[] to C[]
   while (i < na)
      C[k++] = A[i++];

   while (j < nb)
      C[k++] = B[j++];

   nc = k;
}
```

Matrix multiplication

```
#define N 100
typedef int Matrix[N][N];  //fixed-size matrixes

//compute C = A x B
void matrixMul(Matrix A, Matrix B, Matrix C)
{
   for (int i = 0; i < N; i++)
   {
      for (int j = 0; j < N; j++)
      {
         C[i][j] = 0;

         for (int k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
      }
   }
}
```

Pseudo Code

We need a language to express program development

- English is too verbose and imprecise.
- The target language, e.g. C/C++, requires too much details.

Pseudo code resembles the target language in that

- it is a sequence of steps (each step is precise and unambiguous)
- it has similar control structure of C/C++

Pseudo code

```
x = max{a, b, c}
```

C code

```
x = a;
if (b > x)    x = b;
if (c > x)    x = c;
```

Pseudo code allows the programmer to concentrate on problem solving instead of code generation.

Example: Saddle point in 2-D matrix

An *m*×*n* matrix is said to have a saddle point if some entry A[i][j] is the smallest value on row *i* and the largest value in column *j*.

```
An 6×8 matrix with a saddle point
11 33 55 16 77 99 10 40
29 87 65 20 45 60 90 76
50 53 78 44 60 88 77 81
46 72 71 23 88 26 15 21
65 83 23 36 49 57 32 14
70 22 34 19 54 37 26 93
```

Problem:
Given an *m*×*n* matrix, determine if there exists one or more saddle points.

Solution expressed in pseudo code:

```
for each row
{
   j = index of the smallest element on current row, i;

   if (A[i][j]) is the largest element in column j)
      A[i][j] is a saddle point;

}
```

```
Refined pseudo code

for (i = 0; i < m; i++) //for each row
{
   j = index of smallest element on row i;

   //assert: A[i][j] is the smallest element on row i

   for (k = 0; k < m; k++)  //for each element in column j
      if there does not exist A[k][j] > A[i][j]
         A[i][j] is a saddle point;

}
```

```
/* precondition: elements in a row are distinct.
   m = no. of rows
   n = no. of columns  */
void SaddlePoint(int *A[], int m, int n)
{
   for (int i = 0; i < m; i++)
   {
      int j=0, k;
      for (k = 1; k < n; k++)
         if (A[i][k] < A[i][j])
            j = k;

      //assert: a[i][j] is the smallest element on row i.

      int sp = 1;
      for (k = 0; k < m && sp; k++)
         if (A[i][j] < A[k][j]) //a[i][j] is not largest
            sp = 0;             //element in col j

      if (sp)
         cout << "saddle point found at row " << i
              << " , col " << j << endl;

   }
}
```

```
// Version 2: do not assume elements in a row are distinct
int isLargestInCol(int *A[], int m, int r, int c)
{
   //precondition: A[r][c] is a valid element
   for (int i = 0; i < m; i++)
      if (A[r][c] < A[i][c])
         return 0;

   return 1;  //A[r][c] is the largest number in col c
}


void SaddlePoint_2(int *A[], int m, int n)
{
   for (int i = 0; i < m; i++)
   {
      int j=0, k;
      for (k = 1; k < n; k++)
         if (A[i][k] < A[i][j])
            j = k;

      //assert: A[i][j] is the smallest element on row i
      //there may exist A[i][k] == A[i][j] where k > j

      k = j;
      while (k < n)
      {
         if (A[i][j] == A[i][k]
             && isLargestInCol(A, m, i, k))
            cout << "saddle point found at row " << i
                 << " , col " << k << endl;

         k++;
      }
   }
}
```

Some suggestions for good programming style:

- use informative and meaningful variable names
- do not use goto statement, especially backward jump
- use single-entry single-exit control blocks, or at most one break statement inside a loop
- avoid ambiguous statements
  e.g., `x[i] = i++;`
- format the source file with proper indentation of statements and align the braces so that the control structures can be read easily
- insert useful comments (i.e. assertions) in the source program
- avoid side effects (see example in next page)
- minimize direct accesses to global variables, especially you should avoid modifying the values of global variables in a function
- always make a planning of the program organization and data structures before start writing program codes
- should avoid using the trial-and-error approach without proper understanding of the problem to be solved

Example Program with Side Effects

In computer science, a function or expression is said to have a side effect if, in addition to returning a value, it also modifies some state or has an observable interaction with calling functions or the outside world.

```
int x;  //global variable

int f(int n)
{
     x += 1;  //side effect: modify the value of x which is
              //not a formal parameter of function f()

     return n + x;
}

int g(int n)
{
     x *= 2; //side effect

     return n * x;
}

void main()
{    int t;
     …

     t = f(1) + g(2);

     //Logically the same as t = g(2) + f(1)
     //but the results will be totally different.
}
```

Algorithm

An algorithm is a finite set of instructions which, if followed, accomplish a particular task. Every algorithm must satisfy the following criteria:

(i) **input:** there are zero or more quantities which are externally supplied.

(ii) **output:** at least one quantity is produced.

(iii) **definiteness:** each step (instruction) must be clear and unambiguous.

(iv) **finiteness:** the algorithm will terminate after a finite number of steps.

(v) **effectiveness:** every step must be sufficiently basic that it can be carried out by a person using only pencil and paper. Each step must also be feasible.

(vi) **proof of correctness**

In addition to the above criteria, we also want to make our algorithm as efficient as possible.

Example: How to divide (assume that the computer does not have the division operation).

Problem specification:
Given integers $M \geq 0$ and $N > 0$, find integers Q and R such that

$$M = QN + R \quad \text{and} \quad 0 \leq R < N$$

Q is the quotient and R is the remainder.

Solution strategy (trivial strategy)
Subtract N from M repeatedly until the subtraction would yield a negative result.

Structured algorithm:

```
/* precondition: given M >= 0 and N > 0  */
R = M;
Q = 0;
while (R >= N)
{   Q = Q + 1;
    R = R - N;
}
/* postcondition: M = QN + R, 0 <= R < N  */
```

- Preconditions specify the domains of the algorithm.
- Postconditions specify the results of the algorithm.
- If the preconditions are satisfied, then the postconditions are guaranteed.

Semi-formal proof of correctness

1. Termination

If N > 0, the loop is guaranteed to terminate within finite time.

2. Correctness

(i) At termination

(a) when the while-loop terminates, R < N

(b) R = M and M >= 0 initially, thus R >= 0 initially

(c) R = R – N is "guarded" by the while condition, thus R cannot become negative.

(ii) M = QN + R is preserved

initially, Q = 0 and R = M

within the loop

Q = Q + 1

R = R – N

let Q' = Q + 1 and R' = R – N

M = QN + R

= (Q' – 1)N + (R' + N)

= Q'N + R'

if M = QN + R is true before the assignments, it will still be true after the assignments.

The equality "M = QN + R" is called the loop invariant.

Loop Design

```
X;
while B  /* I  */
   Y;
```

`X` is a sequence of steps which initializes the loop.

`Y` is a sequence of steps which constitutes the body of the loop.

`!B` is the termination condition

`I` is the loop invariant.

1. Decide what the loop is supposed to accomplish, i.e. the postcondition. Express the postcondition in the form of `I && !B`.

2. Code `X` which establish `I`.

3. Design code for `Y` which must maintain `I` and make progress towards the goal, i.e. to make `B` false.

Example: computation of the GCD of two positive integers

Given two positive integers M and N, and assume M >= N.

M = QN + R, where Q is an integer and 0 <= R < N

Suppose g is the greatest common divisor of M and N.

M/g = QN/g + R/g

Since M can be divided by g, and N can be divided by g,
R/g should also be an integer (that is R is also divisible by g).

```
// given M > 0 and N > 0
m = M;
n = N;

// I ≡ GCD of m and n = GCD of M and N
while ((r = m % n) > 0)
{
   m = n;
   n = r;
}
// postcondition: r = 0 and n is the GCD of M and N
```

Example: exponentiation

Problem specification:
Given two integers M > 0 and N ≥ 0, we want to find $p = M^N$.

Navie implementation:

```
p = 1;
for (i = 1; i <= N; i++)
   p *= M;
```

This method does not work in some application, e.g. cryptography.
Raising a number to its power is useful in many cryptographic applications.
Given positive integers *M*, *N*, and *B*, we want to compute $z = M^N \bmod B$.

M is an 32-bit integer
N is an integer with 30 digits or more
B is an 32-bit integer

A practical solution method to compute exponentiation

Invariant:
Observe that if `y` is even, $x^y = (x^2)^{y/2}$.

```
/* given M > 0 and N >= 0 */
x = M;
y = N;
p = 1;
/* I ≡ p·x^y = M^N */
while (y > 0)
{
     while ((y % 2) == 0)  // y is even
     {
          x = x * x;
          y = y / 2;
     }
     y--;
     p = p * x;
}
/* y = 0 && p = M^N */
```

Space and time complexity

Given a program or an algorithm, we want to estimate how much time and memory space are required to run the program in terms of the size of the problem instance.

Big O-notation

We say that a function $f(n)$ is of the order of $g(n)$, $f(n) = O(g(n))$, then there exists two constants c and n_0 such that $f(n) \leq c \times g(n)$ for all $n > n_0$

Examples:

$$2n^3 + 55n^2 - 18n = O(n^3)$$

$$a^n + n^k = O(a^n), \text{ for } a > 1$$

$$c = O(1)$$

If $f1(n) = O(g(n))$ and $f2(n) = O(g(n))$, then $f1(n)$ and $f2(n)$ belongs to the same complexity class. However, it does not imply that $f1(n) = f2(n)$.

Lower bound:

$$f(n) = \Omega(g(n)) \Rightarrow f(n) \geq c \times g(n) \text{ for } n > n_0$$

Exact bound:

$$f(n) = \Theta(g(n)) \Rightarrow c1 \times g(n) \leq f(n) \leq c2 \times g(n) \text{ for } n > n_0$$

Example: Matrix addition

The problem size is given by N, the dimension of the matrix.

```
     #define N 10
     typedef int Matrix[N][N];

1    void MatrixAdd(Matrix A, Matrix B, Matrix C)
2    {
3
4       for (int i = 0; i < N; i++)
5          for (int j = 0; j < N; j++)
6             C[i][j] = A[i][j] + B[i][j];
7       return;
8    }
```

line	step count	frequency
1	0	0
2	0	0
3	0	0
4	1	N+1
5	1	N(N+1)
6	1	N^2
7	1	1
8	0	0

total = $2N^2 + 2N + 2$

= $O(N^2)$

Sequential search

```
//A[] is an integer array (no ordering is assumed)
//determine if x is contained in A[0..N-1]

found = 0;  //found is false
for (i = 0; i < N && !found; i++)
   if (A[i] == x)
      found = 1; //found is true, terminate the loop,
                 //x == A[i]

//NOTE that i++ before the for-loop is terminated,
//hence, if found == 1, x == A[i-1]
```

Alternatively implementation using a break-statement

```
for (i = 0; i < N; i++)
   if (A[i] == x)
      break;

//if i < N, x == A[i]; otherwise x is not found
```

It would be better to use a while-loop

```
found = 0;  //found is false
i = 0;
while (i < N && !found)
   if (A[i] == x)
      found = 1;
   else
      i++;

//assert: found == 1 && x == A[i]
//        OR found == 0 and x is not contained in A[]
```

Time complexity of sequential search is $O(N)$, where N is the length of the array.

Binary search

```
//A[] is an integer array sorted in ascending order
//determine if x is contained in A[0..N-1]

low = 0;
high = N-1;
found = false;

while (low <= high && !found)
{
   mid = (low + high) / 2;

   if (A[mid] < x)
      low = mid+1;
   else if (A[mid] > x)
      high = mid-1;
   else
      found = true; // x == A[mid]
}
```

Number of loop executions

≤ number of times N can be divided in half with a result ≥ ½

= number of times to double 1 to reach a value > N

= k where $2^{k-1} \le N < 2^k$

= $\lceil \log_2(N+1) \rceil$

= $O(\log_2 N)$

Time complexity of binary search is $O(\log N)$.

Some important complexity classes

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
0	1	0	1	1	2	1
1	2	2	4	8	4	2
2	4	8	16	64	16	24
3	8	24	64	512	256	40320
4	16	64	256	4096	65536	2.09×10^{13}
5	32	160	1024	32768	4294967296	2.6×10^{35}
6	64	384	4096	262144	1.84×10^{19}	1.27×10^{89}

Complexity class		Example applications
constant time	$O(1)$	random number generation, hashing
logarithmic	$O(\log n)$	binary search
linear	$O(n)$	sum of a list, sequential search, vector product
log-linear	$O(n \log n)$	fast sorting algorithms
quadratic	$O(n^2)$	2D matrix addition, insertion sort
cubic	$O(n^3)$	2D matrix multiplication
exponential	$O(a^n)$, $a > 1$	traveling salesman, placement and routing in VLSI, many other optimization problems
factorial	$O(n!)$	enumerate the permutations of n objects

Practical problem sizes that can be handled by algorithms of different complexity classes

Complexity function	Maximum n
1	Unlimited
$\log n$	Effectively unlimited
n	$n < 10^{10}$
$n \log n$	$n < 10^8$
n^2	$n < 10^5$
n^3	$n < 10^3$
2^n	$n < 36$
$n!$	$n < 15$