Data Structures and Algorithms

Lecturer: Dr. Derek Pao
Office: Room P6518
Email: itdpao@gmail.com

Tentative syllabus

- Review of basic concepts
  - scalar variables, pointer, reference variables
  - parameter passing in function call
  - memory allocation/deallocation
  - char, cstring, C-library functions
  - 1-D and 2-D arrays, array mapping function
  - loop design
  - computation complexity
  - C++ class design
    - struct and class
    - constructor and destructor
    - operator overloading, friend function
    - class design with/without dynamic memory allocation
    - object variable vs object reference (pointer)
    - generic programming, template class/functions
    - static and dynamic binding of function calls
- Recursion
- Sorting and Searching
- Linked lists
- Stacks and Queues
- Binary trees
- Hash table
- Graphs (if time allows)

Assessment:

Assignments:	25% (3 to 4 assignments)
Test:	25% (2 tests, 13 Oct and 24 Nov)
Examination:	50%

You can bring lecture notes and personal notes (hardcopy) to the tests and exam.

Assignment submission:

- Upload your program file to Canvas before the deadline.
- Late submissions or submission by other means are not accepted.
- Submitted programs will be compiled and tested using Visual Studio.
- Download and install Visual Studio (free) in your personal computer https://visualstudio.microsoft.com/vs/community

Grading of assignments, test and examination

- Correctness (assignment programs with compilation errors will receive no more than 30% marks)
- Efficiency (time and space)
- Readability, Clarity and Simplicity
- Applicability to general cases

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.
- If a student is found to have copied from others (or let others to copy his program) more than once, all his assignments will get ZERO mark.

Intended Learning Outcomes

Students should be able to

- apply the structural and object-oriented programming approaches to solve computation problems
- solve computation problems using recursion where appropriate
- demonstrate applications of standard data structures such as linked list, stack, queue and tree
- apply different sorting and searching algorithms

References:

C/C++ on-line reference, https://cplusplus.com/reference/

D. S. Malik, Data Structures Using C++, most recent edition.

YouTube channel:
電腦編程教室 Basic Concepts in Computer Programming (Cantonese)
https://youtu.be/PuXPEJ2WhbE

6 Levels of Thinking Every Student Must Master
www.youtube.com/watch?v=1xqerXscTsE

	Level	Learning/Thinking methods	Results/Abilities
Higher levels	6	Create	Hypothesize, expand new knowledges
	5	Evaluate	Prioritize, make judgement, abstraction/conceptualization
	4	Analyze	Generalization, association of various concepts, solve more complex problems
Lower levels	3	Apply	Simple problem solving, plug-and-play type applications
	2	Understand	Explain/comprehend Ask and answer questions on why and why not
	1	Memorize	Remember

Students/graduates that can only master lower levels thinking will likely be replaced by Artificial Intelligence in the near future.

Students should try to attain higher levels thinking in university education.

There is no easy short-cut to learn/acquire higher levels thinking.
One may say that trainings to acquire higher levels thinking are often "uncomfortable" and even stressful.

This course is not suitable to student who
- is not interested in programming (problem-solving)
- is not willing to spend hours to work on the tutorials/assignments
- relies solely on memorizing facts in his/her studies
- only wants to get a good grade on the transcript without paying the efforts (skipping classes, not doing the assignments by himself/herself).

Different aspects of programming skills:

Coding

- You have a solution strategy. You can work out the answer using paper and pencil without difficulty.
- You should be able to organize and express the computation steps using the target programming language in a clear and precise manner.
- You should train yourself to read and understand program codes. Also learn from the coding style of the example programs in the notes and tutorials.

Debugging

- This is the most painful part in software development.
- Try to avoid errors in the first place → minimize the time for debugging.
- You need to have a good understanding of the algorithm (solution strategy), the programming language and system specific properties.
- You need to pay attention to details in order to spot the errors.

Algorithm design

- This is the ability to solve more complex problems. This will often require domain specific knowledges.
- This is the ability to think in abstract terms, and to see patterns and draw inferences from observations.
- Typically, you need to observe and make use of suitable properties of the data structures in order to derive an effective solution.
- There is no universal strategy that can solve all problems.
- In this course, I shall introduce to you how to design an algorithm for some classical problems. You should pay attention to the design process rather than memorizing the program codes. Questions in the test/exam will NOT be identical to the examples used in the notes or tutorials.

Proper approach to design computer program

Steps:

1. Study the problem: understand the computation requirements and visualize the computation process (solution strategy).

   Work out a few examples based on your solution strategy using paper and pen.
2. Make a plan for the program organization, function interfaces, control block structures (loop control), and the required variables.
3. Choose meaningful (easy to understand) names for variables and functions. In general, name of a function represents the action or result to be produced. Name of a variable is usually a noun that describes its meaning.
4. Design a draft in the form of pseudo-codes using paper and pen.
5. Start writing/typing program statements only when you have an overall picture in your mind.
6. Examine the program carefully, and try to make sure that it is correct.
7. Only when you are convinced that the program is correct, then test the program with properly designed test data.
8. After you get the correct outputs, try to further improve and simplify your program.

Goals:
No need to do any debugging (except for typo errors) for simple problems (e.g. tutorial exercises).

If errors are observed, go back to examine the program codes.

Ability to locate the source of an error is an important skill that you need to master.

Fix the errors and learn from the mistakes (do not repeat the same mistake in the future).

General remarks on program (or algorithm) design

- Explicit (meanings of statements are clear and easy to understand) is better than implicit (statements with hidden assumptions, or meanings not apparent).
- Simple is better than complex.
- Readability counts. Proper comments/assertions that explain the intention of the program statements are helpful.
- Avoid errors in the first place → minimize the time for debugging.
- If the implementation is hard to explain, it is a bad idea (even if the program is correct).
- If the implementation is easy to explain, it may be a good idea.
- Complicated program for solving a simple problem is generally not acceptable.
- Ad hoc and unstructured program that works on the small scale is often a problem in large systems.

Data versus value

$$\begin{tikzcd}[column sep=huge] \text{Data (bits/bytes in memory)} \arrow[r, shift left, "\text{interpretation}"] & \text{Value (abstract concept)} \arrow[l, shift left, "\text{representation}"] \end{tikzcd}$$

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)

Interpretation of a data item can be built-in the system

- hardware (ALU within the CPU)
- language / compiler

```
Example:

char c = 'A';     // bit pattern stored in c is 0100 0001

cout << c << endl;       // what are the outputs ?
cout << c + 1 << endl;
```

Other interpretations of a data item may be imposed by the programmer, e.g. an unsigned int can be interpreted as a 32-bit IPv4 address.

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

- Interpretation is built-in the hardware circuits of the CPU or compiler

data type	size (byte)	interpretation/representation	range of values
bool	1	Boolean	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	$\pm1.4\times10^{-45}$ to $\pm3.4\times10^{38}$
double	8	IEEE 64-bit floating point number	$\pm5\times10^{-324}$ to $\pm1.798\times10^{308}$
pointer	4	memory address	0 to $2^{32}-1$

Remark:
In C/C++, a non-zero integral value (`char, int, short, long, long long, pointer`) is considered to be "true" in a predicate.

Example:

```
int x;
. . .   // other statements

if (x)  // equivalent to if (x != 0)
```

Operators in C++ (not a complete list)

Operator	Symbol	Description
Assignment	=	store a value into a variable
Arithmetic	+, -, *, /, %	
Increment, decrement	++, --	
Unary minus	-	negation
Relational	==, !=, <, <=, >, >=	equal, not equal, less than, etc.
Logical	&&, ||, !	and, or, not
Bitwise	~, &, |, ^, <<, >>	complement, and, or, exclusive-or shift left, shift right
Insertion Extraction	ostream **<<** s istream **>>** i	insertion to an output stream, extraction from an input stream
Member and pointer	x[i]	subscript operator x is an array or a pointer
	*x	indirection, dereference operator x is a pointer
	&x	reference (address-of operator) address of x
	x->y	Structure dereference (arrow operator) x is a pointer to object/struct, member y of object/struct pointed to by x
	x.y	Structure reference (field select operator) x is an object/struct, member y of x

Scalar variables and pointers

Example 1

```
int a, b, c;  // scalar variables
int *p;   /* p is an integer pointer, data type is int*
             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 = nullptr 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
```

The NULL value in C/C++ is used to denote an invalid address.

The symbol NULL represents the value 0 (the compiler replaces NULL with 0 when translating the program statements to machine codes).

```
int *p = NULL;
double *q = NULL;
// Assign a NULL value to a pointer means that the pointer
// variable is not pointing to a valid data item.

int i = NULL;      // same effect as i = 0
double d = NULL;
// DO NOT do this in your program design.
// Syntactically correct, but the semantics (meaning) of
// the statement is confusing. Zero is not an invalid value.

// correct way to initialize a scalar variable
int i = 0;
double d = 0;
```

In the current version of C++ it is preferred to use `nullptr` instead of `NULL`.

```
int *p = nullptr;   // OK

int i = nullptr;  // syntax error, i is not a pointer
```

Parameter passing in function call

1. Pass by value

```
void swap1(int x, int y) // x and y are input parameters
{
    int t;  // t is a local (automatic) variable

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

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

     swap1(i, j);
     cout << "i = " << i << ", j = " << j << endl;
    // output: 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 swap2(int *x, int *y) // x and y are integer pointers
{
     int t;

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

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

     swap2(&i, &j); // pass the address of i and j to swap()
     cout << "i = " << i << ", j = " << j << endl;
    // output: i = 2, j = 1
}
```

3. Pass by reference in C++ (NOT supported in a pure C language environment)

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

```
void swap3(int& x, int& y) //x and y are passed by reference
{
    int t;

    t = x;  // x references i
    x = y;  // y references j
    y = t;
}

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

    swap3(i, j);
    cout << "i = " << i << ", j = " << j << endl;
    // output: i = 2, j = 1

    // Compiler finds that swap3() receives the arguments
    // by reference, the references (addresses) of i
    // and j are passed to swap3().
    // After the function return, i = 2, j = 1.
}
```

Reference data type

```
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;   // &i means address of i

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

```
Outputs produced:

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

```
int i;
int& r = i;

r = 4;    //r references i, value of i is changed to 4
int *q;
q = &r;   //q points to i !!

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

```
Outputs produced:

Execute r = 4
value of i = 4
value of *q = 4
```

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

- It is not possible to refer directly to a reference variable 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.

```
int& r[10]; // array of reference type is 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 block of the class's constructor.

Arrays

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

For example:

```
void myFunction()
{
   int b[10]; // static (compile-time) allocation

   int n = 100;
   int* a = new int[n]; // dynamic (run-time) allocation
                        // of the array

   // other statements in the function
}
```

Variables `a`, `b`, and `n` are called automatic variables. They are mapped to memory locations in the system stack. They cease to exist (destroyed) after the function returns.

The array created by calling the **`new`** operator is allocated in the heap area (a dynamic memory pool managed by the operating system). The memory block is still in use after the function returns.

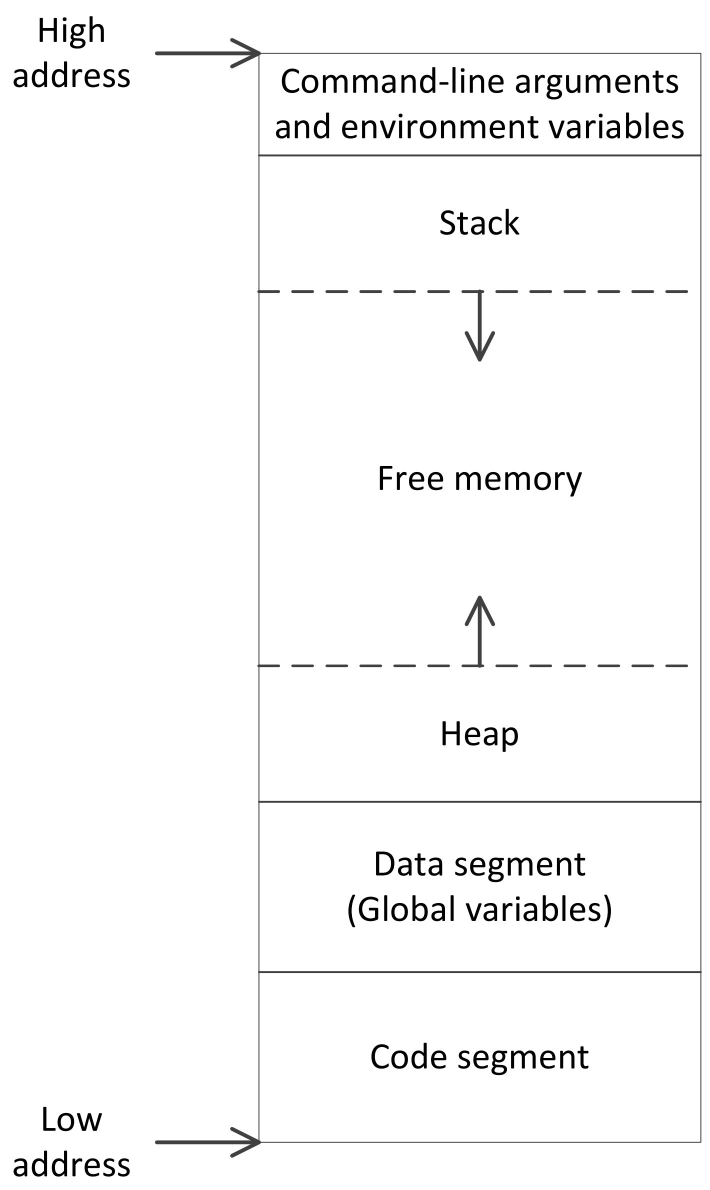
Memory allocation can also be done using the system calls `malloc()` and `calloc()`.
Memory deallocation is done using the `delete` operator or system call `free()`.

Programmers take full control of memory management in C/C++.

Remark:

```
int* p = new int[0]; // create an array of size 0

if (p == nullptr)
   cout << "p is nullptr\n";
else
   cout << "p is not nullptr\n";

// p is nullptr or not ??
```

Memory Layout of C Program


High address
Command-line arguments and environment variables
Stack
Free memory
Heap
Data segment (Global variables)
Code segment
Low address

The stack segment is used to support function calls.
The heap segment is used to support dynamic memory allocation.

Stack and heap segments grow in opposite directions.

When the stack pointer meets the heap pointer, free memory is exhausted and the program will be terminated.

The default amount of free memory is typically a few hundred MBytes in today's computer system.

Array mapping function:

Address of the first element in an array is called the *base address* of the array. Today's computers are byte-addressable (address value refers to a specific byte). Data objects are word-aligned when stored in the memory, e.g. the address of an `int` must be multiple of 4, address of a `double` must be multiple of 8.

Let *esize* be the size of the array element (in term of bytes).

address of $b[i] = base(b) + i * esize$

The size of `int` = 4 bytes.
Index value of an array always starts at 0 (0-based indexing in C/C++).

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

Version 1:

```
int sum_1(int a[], int n)   // n = no. of elements in a[]
{
    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;
}
```

Version 2:

```
// An array is passed to a function by 2 parameters:
// base address (pointer) and size (length of array)
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 is translated to a+i*esize by the compiler
        // 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

void myFunc(int n)
{
   int A[n]; // Error: value of n is not known during
             // compilation time.
   int i, j;

   // function body

}
```

Addition and subtraction operations on pointers

```
double d[10];
double *p, *q, *r;

p = d;     // p points to d[0]
q = p + 8; // q points to d[8]

int k  = q - p; // p and q must be pointer of same data type
// k = 8, number of elements (double) between q and p

p++; // after increment, p points to d[1]
q--; // after decrement, q points to d[7]

r = p + q; // syntax error
// the 2nd operand of the + operator must be int or unsigned
```

Multiplication and division operations on pointers are NOT allowed.

Simple examples on loop design for the processing of arrays.

Typically, the loop test in array processing is used to guard against the index out-of-bound error.

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

int sum = 0;  //initialization
for (int 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:

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

do-loop

```
int sum = 0;
int 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.

Avoid using `!= (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 ?
Debugging of the program for this type of error can be very difficult.

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 body of a for-loop.

```
for (i = 1; i < n; i++) // DO NOT write program in this way
{
   // main body of the loop

   if (someCondition)
      i = i + displacement; // conditional update to i

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

Use a while-loop instead of for-loop if you need to update the loop index conditionally inside the loop.

ASCII table

Hex	Dec	Char		Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
0x00	0	NULL	null	0x20	32	Space	0x40	64	@	0x60	96	`
0x01	1	SOH	Start of heading	0x21	33	!	0x41	65	A	0x61	97	a
0x02	2	STX	Start of text	0x22	34	"	0x42	66	B	0x62	98	b
0x03	3	ETX	End of text	0x23	35	#	0x43	67	C	0x63	99	c
0x04	4	EOT	End of transmission	0x24	36	$	0x44	68	D	0x64	100	d
0x05	5	ENQ	Enquiry	0x25	37	%	0x45	69	E	0x65	101	e
0x06	6	ACK	Acknowledge	0x26	38	&	0x46	70	F	0x66	102	f
0x07	7	BELL	Bell	0x27	39	'	0x47	71	G	0x67	103	g
0x08	8	BS	Backspace	0x28	40	(	0x48	72	H	0x68	104	h
0x09	9	TAB	Horizontal tab	0x29	41	)	0x49	73	I	0x69	105	i
0x0A	10	LF	New line	0x2A	42	*	0x4A	74	J	0x6A	106	j
0x0B	11	VT	Vertical tab	0x2B	43	+	0x4B	75	K	0x6B	107	k
0x0C	12	FF	Form Feed	0x2C	44	,	0x4C	76	L	0x6C	108	l
0x0D	13	CR	Carriage return	0x2D	45	-	0x4D	77	M	0x6D	109	m
0x0E	14	SO	Shift out	0x2E	46	.	0x4E	78	N	0x6E	110	n
0x0F	15	SI	Shift in	0x2F	47	/	0x4F	79	O	0x6F	111	o
0x10	16	DLE	Data link escape	0x30	48	0	0x50	80	P	0x70	112	p
0x11	17	DC1	Device control 1	0x31	49	1	0x51	81	Q	0x71	113	q
0x12	18	DC2	Device control 2	0x32	50	2	0x52	82	R	0x72	114	r
0x13	19	DC3	Device control 3	0x33	51	3	0x53	83	S	0x73	115	s
0x14	20	DC4	Device control 4	0x34	52	4	0x54	84	T	0x74	116	t
0x15	21	NAK	Negative ack	0x35	53	5	0x55	85	U	0x75	117	u
0x16	22	SYN	Synchronous idle	0x36	54	6	0x56	86	V	0x76	118	v
0x17	23	ETB	End transmission block	0x37	55	7	0x57	87	W	0x77	119	w
0x18	24	CAN	Cancel	0x38	56	8	0x58	88	X	0x78	120	x
0x19	25	EM	End of medium	0x39	57	9	0x59	89	Y	0x79	121	y
0x1A	26	SUB	Substitute	0x3A	58	:	0x5A	90	Z	0x7A	122	z
0x1B	27	FSC	Escape	0x3B	59	;	0x5B	91	[	0x7B	123	{
0x1C	28	FS	File separator	0x3C	60	<	0x5C	92	\	0x7C	124	|
0x1D	29	GS	Group separator	0x3D	61	=	0x5D	93	]	0x7D	125	}
0x1E	30	RS	Record separator	0x3E	62	>	0x5E	94	^	0x7E	126	~
0x1F	31	US	Unit separator	0x3F	63	?	0x5F	95	_	0x7F	127	DEL

White space characters include Space, Line feed, Carriage return, Horizontal tab, Vertical tab, and Form feed.

Space = `' '  (0x20)`
Horizontal tab (HT) = `'\t'  (0x09)`
Line feed (LF) is often called newline = `'\n'  (0x0A)`
Carriage return (CR) = `'\r'  (0x0D)`
ASCII codes for `'0'` to `'9'` are 0x30 (hexadecimal) to 0x39.
ASCII codes for `'A'` to `'Z'` are 0x41 to 0x5A.
ASCII codes for `'a'` to `'z'` are 0x61 to 0x7A.

ASCII codes for lowercase and uppercase letters differ by 1 bit (5-th bit).

```
'A'  = 0100 0001
'a'  = 0110 0001
```

```
char ch = 'A' + 32; // ch = 'a'
```

A cstring is an array of char terminated by the delimiter '\0'.

```
char t[] = "This is a cstring"; // '\0' is added by compiler
```

Contents of the array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t[]	T	h	i	s	*sp*	i	s	*sp*	a	*sp*	c	s	t	r	i	n	g	\0
(hex)	54	68	69	73	20	69	73	20	61	20	63	73	74	72	69	6E	67	00

```
sp denotes the space char ' ' (0x20).
'\0' is the end of string delimiter.
```

Example cstring functions (the actual implementations in the C-library may be different from the codes shown below)

```
unsigned strlen(const char *str)
{
   // return the length of the string (exclude '\0')
   unsigned i = 0;
   while (str[i] != '\0')
      i++;

   return i;
}

// Only need to provide the base address when passing
// a cstring to a function.
int strcmp(const char *str1, const char *str2)
{
   // compare str1 with str2
   int diff = 0;
   for (int i = 0; diff == 0 &&
                  (str1[i] != '\0' || str2[i] != '\0'); i++)
      diff = str1[i] - str2[i];

   return diff;

   // if str1 > str2,  return a positive value
   // if str1 == str2, return 0
   // if str1 < str2,  return a negative value

   // Remark: implementation of the C-library function
   //         returns 1, 0, or -1
}
```

```
char* strchr(char *str, int c)
{
   //The parameter c (search char) is a 4-byte int

   //return a pointer to the first occurrence of character c
   //in the cstring str
   //'\0' is considered part of the cstring in this function

   // assert the search char can be represented in 1 byte
   assert(c >= -1 && c <= 255);

   char* p = str;
   while (p != nullptr && *p != c)
      p = (*p != '\0') ? p+1 : nullptr;

   return p;
}
```

```
char* strstr(char *str1, const char *str2)
{
   // Return the address at which the substring str2 is
   // found in str1 (using case-sensitive comparison).

   // Actual implementation in C library may not be the
   // same as the program shown here.

   for (char* p = str1; *p != '\0'; p += 1)
   {
      bool match = true;
      for (int i = 0; match && str2[i] != '\0'; i++)
         match = p[i] == str2[i];
         // no index bound check: is p[i] valid ?
      if (match)
         return p;
   }
   return nullptr;

   // Time complexity of this design is O(nm)
   //    n = length of str1, m = length of str2

   // Remark: a more sophisticated algorithm can achieve
   //         sub-linear time complexity for average case,
   //         i.e. less than O(n) time.
}


// Compare the above implementation with this alternative
// design

char* strstr(char *str1, const char *str2)
{
   for (int j = 0; j < strlen(str1); j++)
   {
      bool match = true;
      for (int i = 0; match && i < strlen(str2); i++)
         match = p[j+i] == str2[i];

      if (match)
         return &str1[j];
   }
   return nullptr;
}
```

Remark:
There are 2 versions of the functions `strchr()` and `strstr()` in the C-library

```
char* strchr(char *str1, int c)
const char* strchr(const char *str1, int c)

char* strstr(char *str1, const char *str2)
const char* strstr(const char *str1, const char *str2)
```

What is the difference between the 2 versions?

Which version is used when you invoke the function in your program statement?

```
int atoi(const char *a)  // ASCII array to integer
{
   // Convert a string of digits (base 10, sign-magnitude)
   // to an integer (2’s complement).

   int i = 0;
   while (isspace(a[i])) // skip leading white space char
      i++;

   int sign = a[i] == ‘-’ ? 1 : 0;

   if (a[i] == ‘-’ || a[i] == ‘+’)
      i++;

   int value = 0;
   while (isdigit(a[i]))  // looping stop when seeing ‘\0’
   {                      // or a char that is not a digit
      value = value * 10 + (a[i] – ‘0’);
      i++;
   }

   if (sign)
      value *= -1;

   return value;
}
```

Commonly used C-library functions:

```
int isspace(int c) : test if c is white-space char

int isdigit(int c) : test if c is a digit ‘0’ to ‘9’

int isalpha(int c) : test if c is a letter ‘A’ to ‘Z’ or
                     ‘a’ to ‘z’

int isalnum(int c) : test if c is a letter or digit
```

Remark:

```
if (isspace(c) == true) // Incorrect usage, why ?
    action;

if (isspace(c))         // Correct
    action;
```

```
unsigned wordCount(const char *a)
{
   // Count the number of words in a[].
   // A word is a consecutive sequence of non-white space
   // chars delimited by white-space char or '\0'.

   /* General loop design consideration:

      inspect each char in a[] iteratively

      if (the char is white-space char)
         what to do
      else // the char is non-white space char
         what to do
   */

   unsigned count = 0;
   bool isWS = true;    // is white-space

   for (int i = 0; a[i] != '\0'; i++)
   {
      if (isspace(a[i]))
         isWS = true;
      else
      {
         if (isWS)    // transition from white-space to
            count++; // non-white space char
                     // a[i] is first char of a word
         isWS = false;
      }
   }
   return count;
}
```

Examples on 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;   // incorrect initialization

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

   return max; //wrong result if numbers in A[] are negative
}
```

In the following examples, I would like to draw your attention to the uses of

preconditions – properties (requirements) 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
If the array is empty, the default answer is zero (or other constant deemed appropriate in the specific application).

Version 1: unordered array

```
//  precondition: A[] is unordered, A[] may be empty
int findMin_1(int A[], int n)   //n = no. of elements in A[]
{
   int min = n > 0 ? A[0] : 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: A[] is in ascending order, A[] may be empty
int findMin_2(int A[], int n)   //n = no. of elements in A[]
{
   //assert: given A[] is sorted in ascending order
   //        A[0] is the smallest number in A[0..n-1]

   return n > 0 ? A[0] : 0;
}
```

Find the location of the minimum value in an array of integers.
If the array is empty, return -1 to represent an invalid index position.

```
//  precondition: A[] is unordered, A[] may be empty
int findMinLoc_1(int A[], int n)
{
   int min;

   min = n > 0 ? 0 : -1;
   for (int i = 1; i < n; i++)
      //assert: ???
      if (A[i] < A[min])
         min = i;

   return min;  // return -1 if A[] is empty
}
```

```
//  precondition: A[] is in ascending ordered
int findMinLoc_2(int A[], int n)
{
   return n > 0 ? 0 : -1;
}
```

Remove duplicated values in an array of integers

Unordered array

```
A[] = {3, 6, 4, 0, 3, 3, 5, 6, 4, 3, 2, 8}
n = 12

After removing duplicated values
A[] = {3, 6, 4, 0, 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: 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++;   // increment j is conditional
      }
   }
   /* Postcondition: elements in A[0..n-1] are distinct, and
      A[] is unordered. */
}
```

Contents of the array after removal of duplicated values

index	0	1	2	3	4	5	6	7	8	9	10	11
A[]	3	6	4	0	8	2	5	4	4	3	2	8
	← valid data values in array → **n = 7**							← **logically removed** → **(garbage values)**				

Remark: Misconception of students

```
A[i] = NULL;  // effect is A[i] = 0
// NOT a valid way to remove an element in A[]
```

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.
Retained numbers should be packed towards the left hand side of the array.

```
// Precondition: A[] is in ascending ordered
void removeDup_2(int A[], int& n)
{
   int k = 0;
   // k points to the location to store the next distinct
   // elmement.

   int i = 0;
   while (i < n)
   {
      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

      //invariant: k <= i
      A[k++] = A[i];
      i = j;
   }
   n = k; // k = no. of distinct elements retained in A[]
}
```

Merge two sorted arrays of integers

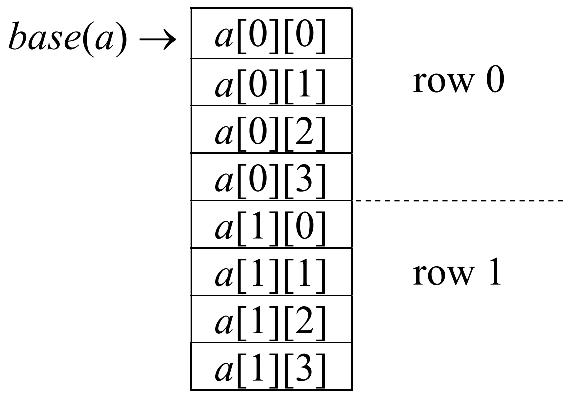
```
//Precondition: A[] and B[] are sorted in ascending order

void merge(const int A[], int na, const int B[], int nb,
           int C[], int& nc) // nc is passed by reference
{
   // array C[] has been created by the calling function

   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 elements in A[] or B[] to C[]
   while (i < na)
      C[k++] = A[i++];

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

   nc = k;
}
```

2D Arrays

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

A multi-dimensional array is 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:

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

base(*a*) = address of *a*[0][0]

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

number of elements placed in front of *a*[i][j] = i × Cols + j

Example: Matrix multiplication

```
#define N 100                // define a constant N = 100
typedef int Matrix[N][N];   // fixed-size 100x100 matrix

// compute C = A × 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];
      }
   }
}
```

Remark:
The above function can only operate on matrix of a fixed size !!

In general, we would like to implement C-functions that can operate on matrixes with different sizes.

To do that, we need another representation of 2D array.

Array of pointers vs 2D-array

```
#define Rows 10
#define Cols 20
int a[Rows][Cols]; // a is a 2D array with 10×20 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: 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:

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

Example:
Function to compute 2D matrix multiplication for matrixes of variable size.

```
// Compute C[n][n] = A[n][n] × B[n][n]
// Precondition: memory space for C[][] has been created by
//               the calling function.

void matrixMul_2(int **A, int **B, int **C, int n)
{
   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 $m \times n$ 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 $m \times n$ matrix, determine if it contains any saddle points.

Solution expressed in pseudo code:

```
for each row (with row index = i)
{
   j = col index of the smallest element on 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
   A[m][n] is a 2D array of integers
*/
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 isSP = 1;  // isSP : is saddle point

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

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

   }
}
```

```
// Version 2: do not assume elements in a row are distinct

//determine if A[r][c] is the largest element in column c
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;
      for (int 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

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

Some suggestions for good programming style:

- format the source file with proper indentation of statements and align the braces so that the control structures can be read easily
- add a space character before and after a binary operator

  Example:

  ```
  // poorly formatted statement
  if(p->sid>q->sid&&p->program!=q->program)

  // properly formatted statement
  if (p->sid > q->sid && p->program != q->program)
  ```

- use informative and meaningful variable and function names
- insert useful comments (i.e. assertions) in the source program
- avoid ambiguous statements
  e.g., `x[i] = i++;`
- minimize direct accesses to global variables, especially you should avoid modifying the values of global variables in a function
- outputs produced by a function should be passed as formal parameters, i.e. avoid unexpected side effects
- use single-entry single-exit control blocks, or at most one `break` statement inside a loop. Avoid the use of `continue` statement.
- do not use goto statement, especially backward jump
- 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 unexpected side effects

```
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;
}

int 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.

General remarks on program (or algorithm) design

- Explicit (meanings of statements are clear and easy to understand) is better than implicit (statements with hidden assumptions, or meanings not apparent).
- Simple is better than complex.
- Avoid errors in the first place, help to minimize time for debugging.
- Readability counts.
- If the implementation is hard to explain, it is a bad idea (even if the program is correct).
- If the implementation is easy to explain, it may be a good idea.
- Complicated program for solving a simple problem is generally not acceptable.
- Ad hoc and unstructured program that works on the small scale is often a problem in large systems.

Some common errors of students:

- When you process an array, be careful about index out-of-bound error,

  i.e. `index < 0` or `index >= size_of_array` (logical size or physical size).

  Error for `index >= logical_size` of an array is more difficult to debug, because the error will only be observed at some later steps of the program.

- When using pointers, be careful about the null-pointer exception,

  i.e. you must make sure that the pointer variable has a valid value before using it to access the data item.

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 $R' = R - N$

$$\begin{aligned} M &= QN + R \\ &= (Q' - 1)N + (R' + N) \\ &= Q'N + R' \end{aligned}$$

    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 establishes `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).

Hence, if R > 0, the GCD of M and N is equal to the GCD of N and R.

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

// I ≡ GCD of m and n is equal to 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 $M > 0$, $N \geq 0$ and $B > 0$, we want to find $p = M^N \bmod B$

Naive implementation:
```
p = 1;
for (i = 1; i <= N; i++)
   p = (p * M) % B;
```

This method does not work in some application, e.g. cryptography.

For example:

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

A practical algorithm to compute exponentiation

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

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

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.

```
1    void MatrixAdd(int **A, int **B, int **C, int n)
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

$$\text{total} = 2n^2 + 2n + 2$$
$$= O(n^2)$$

Sequential search

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

int seqSearch(int A[], int N, int key)
{
  int loc = -1; // location of the element to be searched

  for (int i = 0; i < N && loc < 0; i++)
    if (A[i] == key)
      loc = i;

  return loc;

  // if returned value is < 0, then key is not found
  // otherwise, key == A[loc]
}
```

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 key is contained in A[0..N-1]

int binSearch(int A[], int N, int key)
{
  int low = 0;
  int high = N-1;

  // loop invariant:
  // if key is contained in A[], A[low] <= key <= A[high]
  while (low <= high)
  {
     mid = (low + high) / 2;

     if (key == A[mid])
       return mid;
     else if (key < A[mid])
       high = mid - 1;
     else
       low = mid + 1;
  }

  return -1;
}
// return a negative index : key is not found
// return a non-negative index : key is found at the given
//                               index location
```

Number of loop executions

$\leq$ number of times N can be divided in half with a result $\geq$ ½
= number of times to double 1 to reach a value > N
= k where $2^{k-1} \leq N < 2^k$
= $\lceil \log_2(N+1) \rceil$
= $O(\log_2 N)$

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

See the YouTube video for variants of the binary search that illustrate the application of the concept of loop-invariant in algorithm design.
https://youtu.be/BcNrRx14d00

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, bubble 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

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	$n < 36$
$n!$	$n < 15$