Some Preliminaries and a Quick Review

60-141: Introduction to Algorithms and Programming II

Term: Summer 2013 (July-August)

Instructor: Asish Mukhopadhyay

Lecture Outline

- Algorithms and Programming
- Expressing Complexity : Big-Oh notation
- Some C-facts
- A simple C program
- Another simple C program
- Control flow
- Problem Solving

Algorithms and Programming

- A computer program in any language is a strictly formal way of expressing an *algorithm*
- What's an algorithm ?
  - Informally, it is a step-by-step procedure for solving a computational problem
  - A formal definition needs deeper considerations

Algorithm example

- Computational problem
  - Find the *maximum* of a list of *n* integers
- Algorithm
  - Set the first element as the *maximum*
  - Scan through the remaining elements, comparing each with the maximum, and resetting *maximum* if the element is larger
- A program ?

A simple C Program

```
int main(void)
{
    int list[7] = {2, 4500, 12, 6, 78, 1790, 3};
    int maximum;
    int i; //loop variable

    maximum = list[0];

    for(i = 1; i < 7; i++)
       if (maximum < list[i]) maximum = list[i];

     printf("The maxiumum is: %d", maximum);
     return 0;
}
```

Algorithm Complexity Question

- How many times do we reset the variable *maximum*, if the list is of size *n* ?
  - In the worst-case ?
  - On average ?
- Answers to questions like this are expressed using, what is known as the big-Oh notation

Expressing Complexity : Big-Oh

- Let $f(n)$ and $g(n)$ be non-negative functions of the non-negative integer n. Then,

  $$f(n) = O(g(n))$$

  is shorthand for

  $$f(n) \leq c*g(n)$$

  for $n \geq n_0$ and $c > 0$

Graphical Illustration

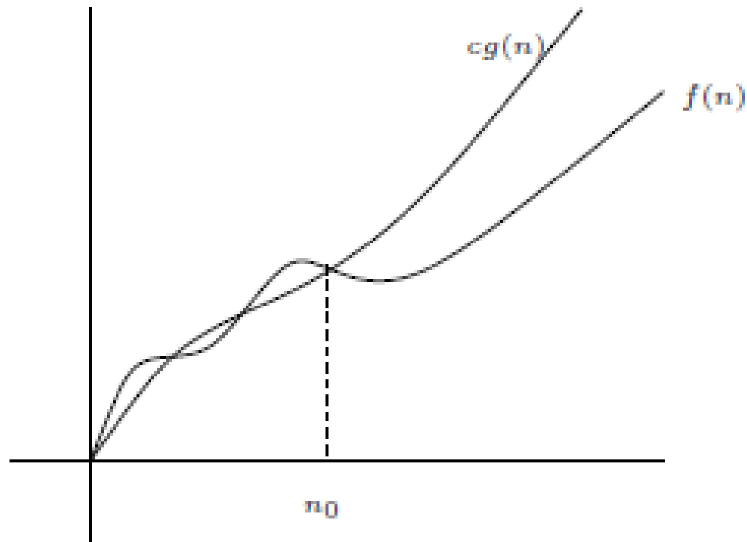


Figure 2.1: Graphical illustration of $f(n) = O(g(n)$

Learning tip

- Big-Oh means *ceiling* or *upper bound*

Notion of a dominant term

- Let n be a positive integer. Consider the function

$$f(n) = 2n^3 + n^2 + n + 1$$

- $2n^3$ determines how $f(n)$ increases as n grows large

Growth Table

n	1	n	n^2	$2n^3$	$f(n)$
10^0	1	1	1	2	5
10^1	1	10	100	2000	2111
10^2	1	100	10000	2000000	2010101
10^3	1	1000	1000000	2000000000	2001 001 001

Formally

- A term *a(n)* of a function *f(n)* **dominates** another term *b(n)*
  - if *b(n)/a(n)* goes to 0 as *n* goes to ∞

Examples

- If $f(n) = \sqrt{n} + \log n + 1$, the term $\sqrt{n}$ dominates both log n and 1

- If $f(n) = 2^n + n^2 + n \log n + 1$, the term 2^n dominates the remaining 3 terms

Reinforce

- Let $f(n) = n^4 + n^3 + n^2 + 1$. Prove formally that $f(n) = O(n^4)$.
- Let $T(n)$ be the running time of the program fragment. Give a big-Oh estimate of $T(n)$.

```
for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
        for(int k = 0; k < j; k++)
        sum++;
```

Running time and Input size

- Given a table like

Input Size	Time
I_1	T_1
I_2	T_2
.	.
.	.
.	.

- Can we find $T = f(I)$?

Too ambitious!

- Instead, try to find a ceiling or upper bound for T

- Mathematically, find $f(I)$ such that
  - $T(I) = O(f(I))$, where I is the input size

Some examples

- *Maximum* of a list of *n* elements
  - $T(n) = O(n)$
- Find a *closest pair* of a set of *n* points
  - $T(n) = O(n^2)$
- Given *n* points in a plane determine if any 3 are *collinear*
  - $T(n) = O(n^3)$

Some C-facts

- C is what is called a high-level programming language
- Evolved from BCPL (Martin Richards, 1967) and B (Ken Thompson, 1970)
- C (Dennis Ritichie, 1970) evolved from B
- Traditional C was expounded in the famous text "*The C Programming Language*" by Kernighan and Ritchie, 1978

Some C-facts

- C programs are compiled and then executed
- The compilation-execution process are shown schematically in the next slide

Compilation-execution process

- **Phase 1**: Programmer creates program using an editor and stores it on disk
- **Phase 2**: Preprocessor program processes the code
- **Phase 3**: Compiler creates object code and stores it on disk
- **Phase 4**: Linker links object code with libraries, creates an executable file and stores it on disk

Compilation-execution process

- **Phase 5:** Loader puts program in memory
- **Phase 6**: CPU takes each instruction and executes it, possibly storing new data as the program executes

A first C Program

```
// A first program in C

#include <stdio.h>

// program execution begins at main

int main(void)
{
    printf("Welcome to C!\n");
    return 0;
} // end function main
```

Program Anatomy

- Comment lines begin with a “//” and are ignored by a compiler
  - Can also use /* */ for multi-line comments
- **Remark in Context**:
  - The fact that compilers ignore comments should not treated as a sign to omit comments from your program. On the contrary, comment your programs carefully. The importance of this aspect of programming cannot be overemphasized

Program Anatomy

- C Preprocessor directive
  - The symbol # begins a C preprocessor directive
    - `#include <stdio.h>` is a directive to include the contents of the header file `stdio.h` in the program
    - This is used by the compiler when compiling calls to the input/output library functions
    - In this case, the `printf()` function

Program Anatomy

- `int main(void)`
  - Every C program must have this function; program execution begins here
  - A function is a generalized *operator* (like '+, *, ..' that takes some arguments, operates on these arguments and returns a value)
  - `int` indicates that the function returns a value of integer type
  - `void` indicates that it needs no arguments in this case!

Program Anatomy

- `printf(("Welcome to C!\n");`
  - This prints `Welcome to C!` and moves the cursor moves to the beginning of a newline
  - `\n` is called an escape sequence that calls for subsequent output, if any, to be printed on a new line (for some common escape sequences see Fig. 2.2 of your text)

Adding computation

```
// Addition program
#include <stdio.h>

// program execution begins at function main
int main(void)
{
    int integer1; // first summand
    int integer2; // second summand
    int sum;      // result

    printf("Enter first integer\n"); //prompt
    scanf("%d", &integer1); //read an integer

    printf("Enter second integer\n"); //prompt
    scanf("%d", &integer2); // read an integer

    sum = integer1 + integer2; // computing sum

    printf("sum is %d\n", sum); return 0;
} // end function main
```

Program Anatomy again!

```
int integer1;
int integer2;
int sum;
```

```
int integer1, integer2, sum;
```

- The above declarations list the variables to be used and their type, integer in this case
- Consequently, memory locations are reserved for storing and accessing their values
- In C, all variables *must be declared* before use
- The declarations can also be placed in a single line

About identifiers

- An *identifier* is any sequence of letters, digits and underscore(_) that does not begin with a digit
- A variable name is any valid identifier
- Variable names are *case-sensitive*

Program Anatomy again!

- `scanf("%d", &integer1); //read an integer`
  - This command reads formatted input from the terminal
  - "%d" is a format string, %d, indicating that the input to be read is a decimal integer
  - The & in &integer1 is an address operator that gives the address of the variable integer1, where the value read is to be stored

Program Anatomy again!

- `sum = integer1 + integer2;`
  - This is an *assignment statement* , so called as the variable `sum` is assigned the sum of the values of `integer1` and `integer2`
  - `=` is called an assignment operator; it's a binary operator because it needs two operands a variable on the left (sum in this case) and an expression on the right

Program Anatomy again!

- `integer1 + integer2`
  - is an arithmetic expression
  - The + is a binary arithmetic operator
  - Other binary arithemetic operators are:
    - * (multiplication) , / (division) , % (modulus)
- To evaluate an arithmetic expression with more than one arithmetic operator we have to follow some rules of precedence among operators

Precedence Rules

- Rules of precedence for arithmetic operators
  - Bracketed expressions are evaluated first
  - Following that, *, / and % operators are applied from left to right
  - Next, the + and − operators are applied, also from left to right
  - Finally, the = operator is applied

Going forward

- Can’t do much with assignment statements alone!
- So, let’s look at a more complex program

Making decisions (1)

```
// Using if statements, relational
// operators, and equality operators
#include <stdio.h>

// program execution begins at function main
int main( void )
{
   int num1; // first number to be read from user
   int num2; // second number to be read from user

   printf( "Enter two integers, and I will tell you\n" );
   printf( "the relationships they satisfy: " );

   scanf( "%d%d", &num1, &num2 ); // read two integers

   if ( num1 == num2 ) {
      printf( "%d is equal to %d\n", num1, num2 );
  } // end if
```

Making decisions (2)

```
if ( num1 != num2 ) {
      printf( "%d is not equal to %d\n", num1, num2 );
   } // end if

   if ( num1 < num2 ) {
      printf( "%d is less than %d\n", num1, num2 );
   } // end if

   if ( num1 > num2 ) {
      printf( "%d is greater than %d\n", num1, num2 );
   } // end if

   if ( num1 <= num2 ) {
      printf( "%d is less than or equal to %d\n", num1, num2 );
   } // end if

   if ( num1 >= num2 ) {
      printf( "%d is greater than or equal to %d\n", num1, num2 );
   } // end if
} // end function main
```

Program anatomy!

- `num1 == num2`
  - The `==` operator checks if the `num1` and `num2` are equal
- `num1 != num2`
  - The `!=` operator checks if `num1` and `num2` are not equal
- `num1 < num2`
  - The `<` operator checks if `num1` is less than `num2`

Program anatomy!

- `num1 <= num2`
  - The `<=` operator checks if `num1` is less than or equal to `num2`
- `num1 > num2`
  - The `>` operator checks if `num1` is greater than `num2`
- `num1 >= num2`
  - The `>=` operator checks if `num1` is greater than or equal to `num2`

Operator Precedence Table

Operator	Associativity
()	Left to right
*, /, %	Left to right
+, -	Left to right
<. <=, >, >=	Left to right
==, !=	Left to right
=	Right to left

More complex decision-making

- *Selective Computation*
  - Consider the following problem:
    - Given three distinct integers a , b , c find their maximum
  - Algorithm ?
    - max(a , b , c) = max(max(a , b), c)

Program

```
int main(void)
{
    int a,b,c; //input values

    printf("Input three numbers: ");
    scanf("%d %d %d", &a, &b, &c);

    if (a > b) {
          if (a > c) printf("The maximum is %d\n", a);
          else printf("The maximum is: %d\n", c);
    }
    else
    {
        if (b > c) printf("The maximum is: %d\n", b);
        else printf("The maximum is: %d\n", c);
    }
}
```

Syntax of `if-else`

```
if (expression)
    statement1
else
    statement2
```

- Expression non-zero means `true`, else `false`
- `else` is optional

Dangling `else` problem

- As `else` is optional, there's syntactic ambiguity if statement$_1$ has the form `if-else` as in example below.
- *Resolution*: associate an `else` with the nearest `if` (shown by indentation)

```
if (n >0)
   if(a>b)
      z = a;
   else
      z =b;
```

else-if

```
int main(void)
{
    int a;

    printf("Input the number :");
    scanf("%d", &a);

    if(a<0) printf("The input is 
negative\n");
    else if(a>0) printf("The input is 
positive\n");
    else printf("The input is zero\n");
}
```

Syntax of `else-if`

```
if(expression)
    statement
else if (expression)
   statement
else if (expression)
   statement
.
.
.
else if (expression)
   statement
else
    statement
```

The `switch` construct

- Problem:
  - Given a piece of text input, count the number of digits, the number of white spaces and the number of characters in the input

Solution using `switch` (1)

```
int main(void)
{
    char c;
    int ndigits, nwhite, nother;

    ndigits = nwhite = nother = 0;

    while ((c = getchar())!= EOF) {
          switch(c){
                    case '0': case '1': case '2': case
'3': case '4': case '5': case '6': case '7': case '8':
case '9':
                         ndigits++;
                         break;
                    case ' ':
                    case '\t':
                    case '\n':
                          nwhite++;
                          break;
```

Solution using `switch` (2)

```
                    default:
                        nother++;
                        break;
                    }
    }
    printf("white space = %d, number of digits = %d, 
other = %d\n", nwhite, ndigits, nother);
}
```

Syntax of the `switch` construct

```
switch(expression) {
case const-expression: statements
                       break;

case const-expression: statements
                       break;
.
.
.
              default: statements
                       break;
}
```

Repeating : the `for`-loop

- **Problem**: Compute the sum
  - 1 + 2 + 3 + …+ 100

Solution

```
int main(void)
{
    int i, sum;

    sum = 0;

    for(i=1; i <= 100; i++)
        sum = sum + i; // or sum +=i;

    printf("The sum of the first 100 
natural numbers is: %d\n", sum);
}
```

`for`-loop syntax

```
for (expression1; expression2; expression3)
```

- One or more of the three expressions may be absent
- Semi-colons *must be present*, however

Solving with a `while`-loop

```
int  main(void)
{
      int i, sum;
      sum = 0; i = 0;

       while(i <= 100) {
           sum = sum + i;  // or sum +=i
            i = i+1; // or i++
        }
        printf("The sum of the first 100
         natural numbers is %d\n", sum);
        return 0;
}
```

`while`-loop syntax

`while(`expression`)`
    statement

Solving with a `do-while`-loop

```
int  main(void)
{
      int i, sum;
      sum = 0; i = 1;

       do {
           sum = sum + i;  // or sum +=i
           i = i+1; // or i++
           } while(i <= 100);

        printf(“The sum of the first 100
         natural numbers is %d\n”, sum);
        return 0;
}
```

`do-while` Syntax

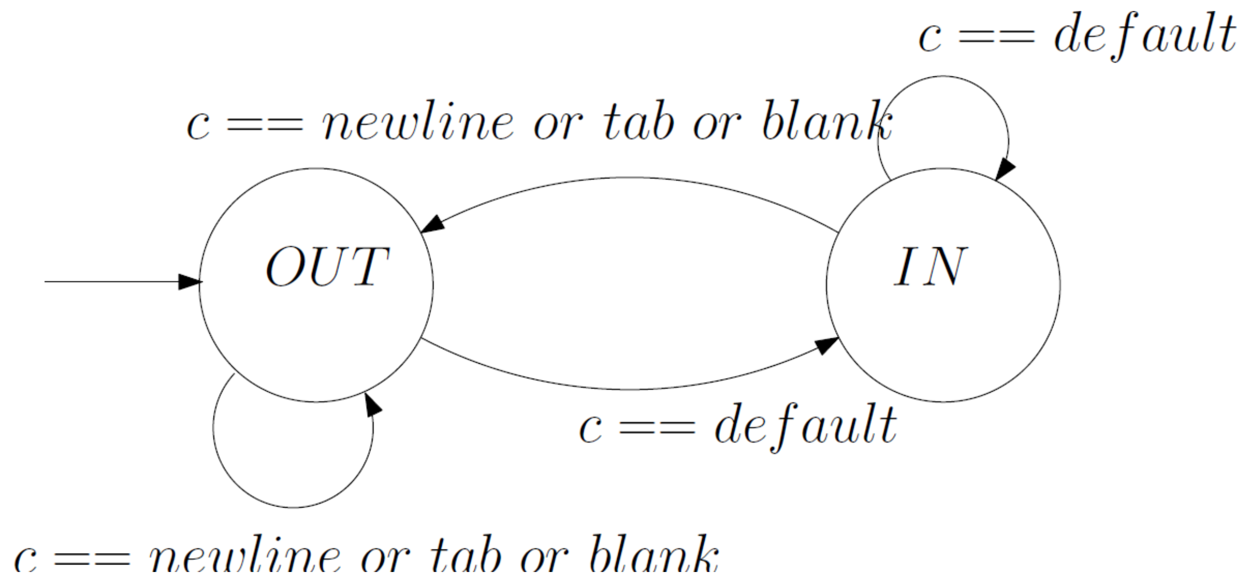
```
do
        statement
while(expression);
```

Class Exercise

- **Problem**:
  - Write a C-program that reads a text input and prints one word per line

Class Exercise 1

- Algorithm



A C-program

```
#include <stdio.h>
#define IN  1
#define OUT 0

main()
{
  int state;
  char c;

  state = OUT;
```

A C-program

```
while((c=getchar())!= EOF)
 {
  switch(c)
  {
    case '\t':
    case ' ' :
    case '\n':
                  if(state == IN)
                 {
                    state=OUT;
                    printf("\n");
                  }
                  break;
    default  : if(state == OUT) state=IN;
                   putchar(c);
                   break;
   }
 }
}
```

Class Exercise 2

- **Problem**:
  - Write a C-program that reads a text input and copies it to the output replacing each string of one or more blanks with a single blank.

A C-program

```
#include <stdio.h>
#define NONBLANK 'a'

/* replace a string of blanks with a
single blank */

int main(void)
{
    int c,
    int lastc; // last character read

    lastc = NONBLANK;
```

A C-program

```
while((c = getchar()) != EOF) {
             if (c!= ' ')
                putchar(c);
             if (c == ' ')
                  if (lastc != ' ')
                      putchar(c);
             lastc = c; // update last char read
    } // end while
} // end main
```