COEN 79
Object-Oriented Programming and Advanced Data Structures
Department of Computer Engineering, Santa Clara University
Behnam Dezfouli, PhD

The Phases of Software Development

Slide Set: 1

Learning Objectives

- Write **precondition/postcondition** contracts for small functions, and use the C++ assert facility to test preconditions
- Recognize quadratic, linear, and logarithmic **running time behavior** in simple algorithms, and write big-*O* expressions to describe this behavior
- Create and recognize test data that is appropriate for simple problems, including testing boundary conditions and fully exercising code

Introduction

The Phases of Software Development

- Specification of the task
- Design of a solution
- Implementation (coding) of the solution
- Analysis of the solution
- Testing and debugging
- Maintenance and evolution of the system
- Obsolescence

The Phases of Software Development

Specification, Design, Implementation

- The **specification** is a precise description of the problem
- The **design** phase consists of formulating the steps to solve the problem
- The **implementation** is the actual C++ code that carries out the design

The Phases of Software Development

Specification, Design, Implementation

- Example: Display a table for converting Celsius temperatures to Fahrenheit:

```
CONVERSIONS FROM -50.0 to 50.0
   Celsius       Fahrenheit
    -50.0C
    -40.0C
    -30.0C
    -20.0C
    -10.0C
      0.0C
     10.0C
     20.0C
     30.0C
     40.0C
     50.0C
```

The actual Fahrenheit temperatures will be computed and displayed on this side of the table.

For a small problem, a sample of the desired output is a sufficient specification

The Phases of Software Development

Specification, Design, Implementation

- The next step is to design a **solution**:
  - An **algorithm** is a set of instructions for solving a problem
  - ❑Example: An algorithm for the temperature problem will print the conversion table
  - During the design of the algorithm, the details of a particular programming language can be distracting
  - Using **a mixture of English and a programming language** (C++) is called **pseudocode**
  - When the C++ code for a step is obvious, then the pseudocode may use C++
  - When a step is clearer in English, then we will use English

The Phases of Software Development

Design Concept: Decomposing the Problem

- A good technique for designing an algorithm is to **break down the problem at hand into a few subtasks**
  - Then decompose each subtask into smaller subtasks
  - Then replace the smaller subtasks with even smaller subtasks, and so forth
  - Each **subtask** can be implemented as a separate piece of your program (C++ **function**)

Design Concept: Decomposing the Problem (Cont'd)

❑Example:

- The temperature problem has at least two good subtasks:
  1. Converting a temperature from Celsius degrees to Fahrenheit
  2. Printing a line of the conversion table in the specified format

Design Concept: Decomposing the Problem (Cont'd)

- The first draft of our **pseudocode** for the temperature problem might look like this:

1. Do preliminary work to open and set up the output device properly
2. Display the labels at the top of the table
3. For each line in the table (using variables `celsius` and `fahrenheit`):
   a. Set `celsius` equal to the next Celsius temperature of the table
   b. `fahrenheit` = the `celsius` temperature converted to Fahrenheit
   c. Print the Celsius and Fahrenheit values with labels on an output line

Design Concept: Decomposing the Problem (Cont'd)

- What makes a good decomposition?
  - Subtasks should help you produce **short pseudocode**
  - No more than a page of succinct description to solve the entire problem, and ideally much less than a page

- Two considerations for selecting good subtasks:
  1. The potential for **code reuse**: A function is written with sufficient generality that it can be reused elsewhere
  2. The possibility of **future changes to the program**
     - Example: Our temperature program might be modified to convert to Kelvin degrees instead of Fahrenheit
     - If the conversion task is performed by a separate function, much of the modification will be confined to this one function

Preconditions and Postconditions

- Frequently a programmer must communicate precisely **what** a function accomplishes, without any indication of **how** the function does its work
- **Information hiding**: To know only as much as you need, but no more

Preconditions and Postconditions (Cont'd)

- You are the head of a programming team and you want one of your programmers to write a function for a part of a project

Preconditions and Postconditions (Cont'd)

- One way to specify such requirements is with a pair of statements about the function
  - The **precondition** statement indicates **what must be true before the function is called**
  - The **postcondition** statement indicates **what will be true when the function finishes its work**

- Precondition and postcondition are added to a function **prototype**
  - Prototype consists of the function's return type, name, and parameter list

Preconditions and Postconditions (Cont'd)

❑ Example

```
void write_sqrt( double x )

// Precondition:  x  >=  0.
// Postcondition: The square root of x has been written to
// the standard output.

  ...
```

The precondition and postcondition appear as comments in program

The Phases of Software Development

Preconditions and Postconditions (Cont'd)

❑ Example

```
void write_sqrt( double x )

// Precondition:  x  >=  0.
// Postcondition:  The square root of x has
// been written to the standard output.

 ...
```

In this example, the precondition requires that `x >= 0` be true whenever the function is called

❑ Which of these function calls meet the precondition?

```
write_sqrt( -10 );
write_sqrt( 0 );
write_sqrt( 5.6 );
```

The Phases of Software Development

Preconditions and Postconditions (Cont'd)

❑ Example

```
void write_sqrt( double x)

// Precondition:  x  >=  0.
// Postcondition:  The square root of x has
// been written to the standard output.


  ...
```

- The postcondition always indicates what work the function has accomplished
- In this case, when the function returns, the square root of `x` has been written

The Phases of Software Development

Preconditions and Postconditions (Cont'd)

Example

```
bool is_vowel( char letter )
// Precondition:  letter is an uppercase or lowercase letter
// (in the range 'A' ... 'Z' or 'a' ... 'z') .
// Postcondition:  The value returned by the function is true if
// Letter is a vowel; otherwise the value returned by the
// function is false.

 ...
```

`is_vowel( 'A' );`	**true**
`is_vowel(' Z' );`	**false**
`is_vowel( '?' );`	**Nobody knows, the precondition has been violated**

Preconditions and Postconditions (Cont'd)

- **The programmer who calls the function is responsible for ensuring that the precondition is valid** when the function is called
- When we pretend that we do not know how a function is implemented, we are using a form of information hiding called **procedural abstraction**

Preconditions and Postconditions (Cont'd)

- The programmer who writes the function counts on the precondition being valid, and **ensures that the postcondition becomes true** at the function's end

Preconditions and Postconditions (Cont'd)

- When you write a function, you should **detect when a precondition has been violated**
- If you detect that a precondition has been violated, then print an error message and halt the program
  - … to avoid crashing the program with an unknown error

Preconditions and Postconditions (Cont'd)

- During program development, **functions should be designed to help programmers find errors as easily as possible**
- As part of this effort, the first action of a function should be to check that its precondition is valid
- If the precondition fails, then:
  - The function prints a message
  - Either halts the entire program, or performs some other error actions before returning
- The **assert** facility is a good approach to detecting invalid data at an early point
  - Includes this directive: `#include <cassert>`
  - You can turn off all assertions by: `#define NDEBUG`

Preconditions and Postconditions (Cont'd)

```
void write_sqrt( double x)

// Precondition: x  >=  0.
// Postcondition: The square root of x has been written
// to the standard output.

{

  assert(x >= 0);

 ...
```

If (x >= 0)
x is valid and the assertion takes no action

If (x < 0)
The precondition has been violated,
a message is printed and the program is halted

The Phases of Software Development

Preconditions and Postconditions (Cont'd)

❑ Example

```
#include <iostream>
// uncomment to disable assert()
// #define NDEBUG
#include <cassert>

int main()
{
    assert(10+10==20);
    std::cout << "Execution continues past the first assert\n";
    assert(12+12==20);
    std::cout << "Execution continues past the second assert\n";
}
```

```
output:
Execution continues past the first assert
test: test.cc:10: int main(): Assertion `12+12==20' failed.
Aborted
```

Preconditions and Postconditions (Cont'd)

- Succinctly describes the behavior of a function...
  - ... without cluttering up your thinking with details of how the function works
- At a later point, you may re-implement the function in a new way ...
  - ... but programs (which only depend on the precondition/postcondition) will still work with no changes

The Phases of Software Development

The Temperature Conversion Program

```
#include <iostream>     // Provides cout
#include <iomanip>      // Provides setw function for setting
                        // output width
#include <cstdlib>      // Provides EXIT_SUCCESS
#include <cassert>      // Provides assert function
using namespace std;    // Allows all standard library items to
                         // be used
```

A function for temperature conversion

```
double celsius_to_fahrenheit(double c)
// Precondition: c is a Celsius temperature no less than
// absolute zero (-273.16).
// Postcondition: The return value is the temperature c
// converted to Fahrenheit degrees.
{
    const double MINIMUM_CELSIUS = -273.16; // Absolute zero in
                                            // Celsius degree

    assert(c >= MINIMUM_CELSIUS);
    return (9.0 / 5.0) * c + 32;
}
```

It is a declared constant, which means that its value will never be changed while the program is running

The Phases of Software Development

The Temperature Conversion Program (Cont'd)

A function to setup the output format

```
void setup_cout_fractions(int fraction_digits)
// Precondition: fraction_digits is not negative.
// Postcondition: All double or float numbers printed to cout
// will now be rounded to the specified digits on the right of the
// decimal.
{
    assert(fraction_digits > 0);

    cout.precision(fraction_digits);

    cout.setf(ios::fixed, ios::floatfield);

    if (fraction_digits == 0)
        cout.unsetf(ios::showpoint);
    else
        cout.setf(ios::showpoint);
}
```

Determines how many digits are printed for each number

Causes numbers to be printed in fixed-point notation

Set the `showpoint` flag to show the decimal point

The Phases of Software Development

The Temperature Conversion Program (Cont'd)

```
int main( )
{
    // Heading for table's 1st column
    const char   HEADING1[]  = "   Celsius";

    // Heading for table's 2nd column
    const char   HEADING2[]  = "Fahrenheit";

    // Label for numbers in 1st column
    const char   LABEL1      = 'C';

    // Label for numbers in 2nd column
    const char   LABEL2      = 'F';

    // The table's first Celsius temp.
    const double TABLE_BEGIN = -50.0;

   // The table's final Celsius temp.
    const double TABLE_END   = 50.0;
```

The main function uses the two functions described earlier

The Phases of Software Development

The Temperature Conversion Program (Cont'd)

```
// Increment between temperatures
const double TABLE_STEP  = 10.0;

const int WIDTH   = 9; // Number chars in output numbers

const int DIGITS  = 1; // Number digits right of decimal pt

double value1;  // A value from the table's first column
double value2;  // A value from the table's second column


// Set up the output for fractions and print the table headings.
setup_cout_fractions(DIGITS);
cout << "CONVERSIONS from " << TABLE_BEGIN << " to " << TABLE_END
                                                    << endl;
cout << HEADING1 << "  " << HEADING2 << endl;
```

Note:
Whenever you write the `endl` object, the output buffer is automatically flushed

The Phases of Software Development

The Temperature Conversion Program (Cont'd)

```
    // Each iteration of the loop prints one line of the table.
    for (value1 = TABLE_BEGIN; value1 <= TABLE_END; value1 +=
                                                    TABLE_STEP)
    {
        value2 = celsius_to_fahrenheit(value1);

        cout << setw(WIDTH) << value1 << LABEL1 << "  ";
        cout << setw(WIDTH) << value2 << LABEL2 << endl;
    }

    return EXIT_SUCCESS;
}
```

`setw(WIDTH)` sets the preferred number of output characters to be used when the item is printed

Use "`EXIT_SUCCESS`" in a Main Program
Same as `return 0`

The Standard Library and the Standard Namespace

- **Standard Library** aids programmers in writing portable code that can be compiled and run with many different compilers on different machines

- To use one of the library facilities, a program places an "include directive" at the top of the file that uses the facility
  - For example, to use the usual C++ input/output facilities: **`#include <iostream>`**
  - Some additional input/output items require a second include directive: **`#include <iomanip>`**

The Standard Library and the Standard Namespace (Cont'd)

Note: Older Names for the Header Files

- Older C++ compilers used slightly different names for header files
- For example, older compilers used `iostream.h` instead of simply `iostream`

The Standard Library and the Standard Namespace (Cont'd)

The Standard Namespace

- All of the items in the new header files are part of a feature called the **standard namespace**, also called **std**
- When you use one of the new header files, your program should also have this statement after the include directives:
  `using namespace std;`
  - Note: There are other alternatives that we will talk about later

The Standard Library and the Standard Namespace (Cont'd)

```
return EXIT_SUCCESS;
```

- Defined in `cstdlib` (or `stdlib.h`)
- The return value of `EXIT_SUCCESS` tells the operating system that the program ended normally, and the operating system can then proceed with its next task
- For most operating systems, this constant is defined as zero (which is why you may have used `return 0` in other programming
- **Another example:**
  - Assume that a process A starts a process B, and waits for the successful completion of B
  - If B does not return an `EXIT_SUCCESS`, then A might try again running this process, or it might simply exit with a failure code

Running Time Analysis

Running Time Analysis

- Time analysis consists of reasoning about an algorithm's speed:
  1. Does the data structure or algorithm work fast enough for my needs?
  2. **How much longer does the method take when the input gets larger?**
  3. Which of several different methods is fastest?

Running Time Analysis

- For a time analysis of a program, **we do not usually measure the actual time taken to run the program**
  - Because the number of seconds can depend on too many extraneous factors (e.g., processor speed, the effect of other processes)

- **The analysis counts the number of operations required**
  - An operation:
    - As simple as the execution of a single program statement
    - Arithmetic operation (addition, multiplication, etc.)
  - For most programs, **the number of operations depends on the input size**
    - For example, a program that sorts a list of numbers is quicker with a short list than with a long list

Big-O Notation

- Often it is enough to know in a rough manner **how the number of operations is affected by the input size**
- We can express this kind of information in a format called **big-O notation** which is the **order of an algorithm**
- The order analysis is often enough to compare algorithms to estimate how running time is affected by changing data size

❑For example, the complexity of insertion into a linked list is O(1), while the complexity of insertion into a sorted array is O(n)

Big-O Notation (Cont'd)

- If the largest term in a formula is a constant times *n* (i.e., $a \times n$), then the algorithm is said to be **"big-O of *n*"** written *O*(*n*), and the algorithm is called **linear**

❑ Example: What is the big-O notation for this formula?
100n+5 : O(*n*)

Big-O Notation (Cont'd)

- If the largest term in a number of operations' formula is no more than a constant times n^2, then the algorithm is said to be "big-O of n^2," written $O(n^2)$, and the algorithm is called **quadratic**

❑ What is the big-O notation for this formula?

n^2+5n : $O(n^2)$

Big-O Notation (Cont'd)

- If the largest term in a formula is a constant times a logarithm of *n*, then the algorithm is "big-O of the logarithm of *n*," written *O*(*log n*), and the algorithm is called **logarithmic**

❑ What is the big-O notation for this formula?
The number of digits in *n* : O(*log n)*

Note: `digits = floor( log10( number ) ) + 1;`

Running Time Analysis

Big-O Comparison of Algorithms

Highest complexity

Lowest complexity

Function	Common name
$n!$	factorial
2^n	exponential
$n^d,\ d > 3$	polynomial
n^3	cubic
n^2	quadratic
$n\sqrt{n}$	
$n \log n$	quasi-linear
n	linear
$\sqrt{n}$	root - n
$\log n$	logarithmic
1	constant

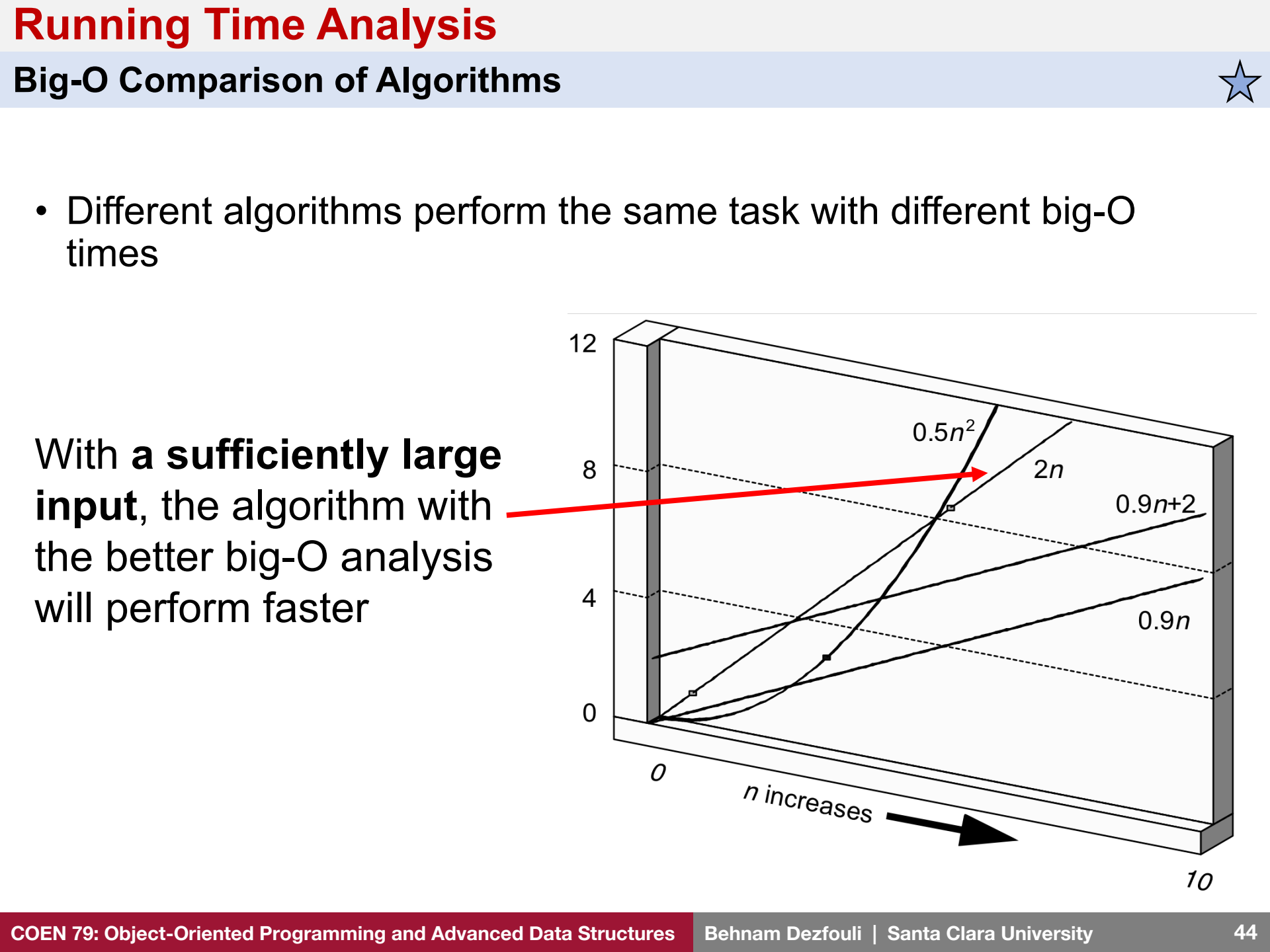
Running Time Analysis

Big-O Notation (Cont'd)

Examples

$T(n)$	Complexity
$5n^3 + 200n^2 + 15$	$O(n^3)$
$3n^2 + 2^{300}$	$O(n^2)$
$5\log_2 n + 15\ln n$	$O(\log n)$
$2\log n^3$	$O(\log n)$
$4n + \log n$	$O(n)$
2^{64}	$O(1)$
$\log n^{10} + 2\sqrt{n}$	$O(\sqrt{n})$
$2^n + n^{1000}$	$O(2^n)$

Big-O Comparison of Algorithms

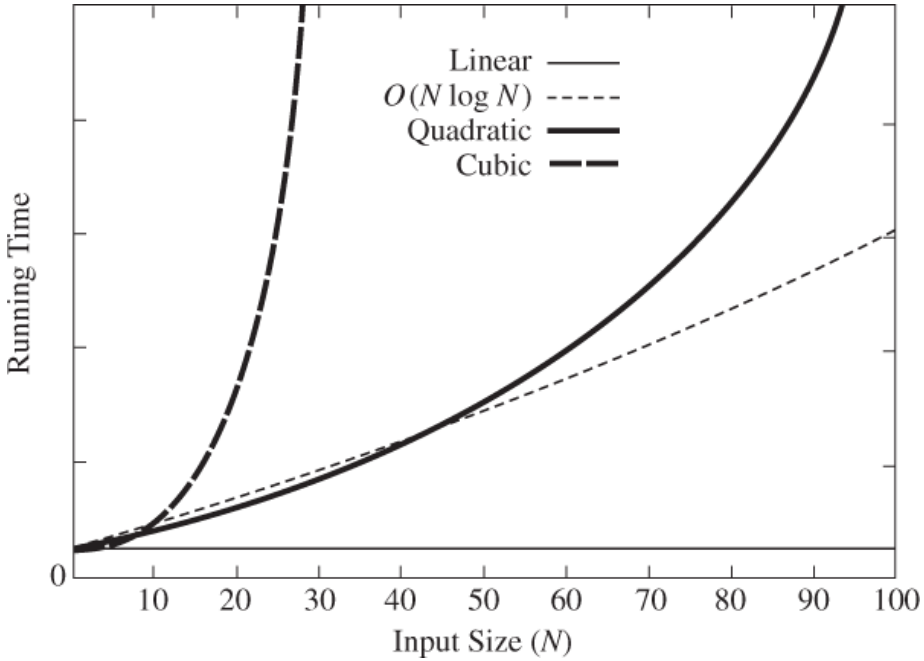


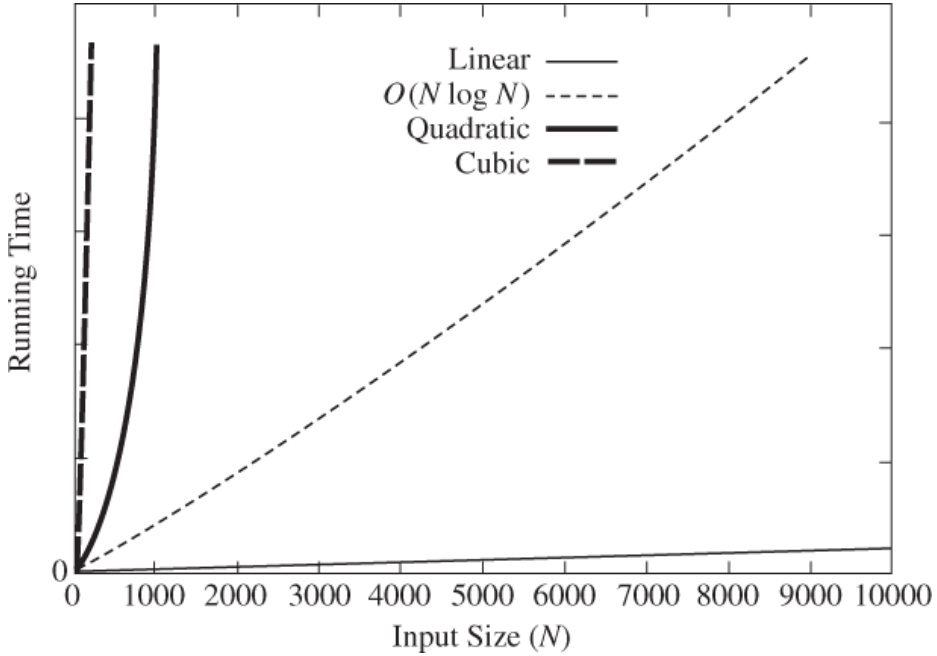
- Different algorithms perform the same task with different big-O times

With **a sufficiently large input**, the algorithm with the better big-O analysis will perform faster

Running Time Analysis

Big-O Comparison of Algorithms



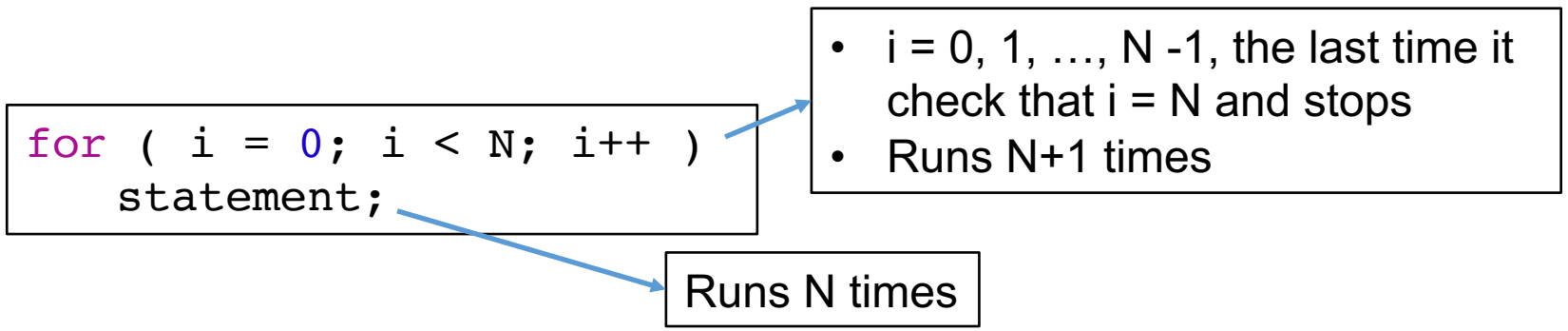


Time Analysis of C++ Functions

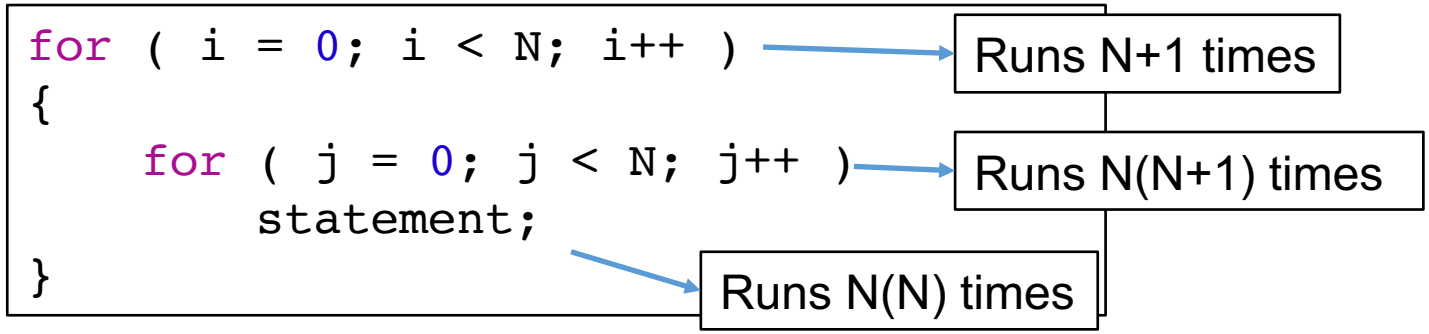
- The first step of the time analysis is **to decide precisely what we will count as a single operation**
  - For C++ functions, a good choice is to count the total number of C++ operations (such as an assignment, the < operation, or the << operation) plus the number of function calls (such as the call to assert)
  - If the function calls did complex work themselves, then we would also need to count the operations that are carried out there

Running Time Analysis

Big-O Notation (Cont'd)



- Total number of operations = N + (N + 1)
- The complexity of this code is O(N)



- The complexity of this code is $O(N^2)$

Big-O Notation (Cont'd)

```
for (int i = 1; i <= n; ++i)
{
    for (int j = 1; j <= i; ++j)
        statement;
}
```

- What is the time complexity of the above code?

Running Time Analysis

- **Worst-case analysis:** determines the maximum number of operations required for the inputs of a given size *n*
- **Average-case analysis**: determines the average number of operations required for the inputs of a given size *n*
- **Best-case analysis**: determines the fewest number of operations required for the inputs of a given size *n*

Running Time Analysis

Sample Actual Execution Time

- If an operation can be run in 1 ns, then the execution time of different algorithms against data size is:

	Time		
Function	$(n = 10^3)$	$(n = 10^4)$	$(n = 10^5)$
$\log_2 n$	10 ns	13.3 ns	16.6 ns
$\sqrt{n}$	31.6 ns	100 ns	316 ns
n	1 μs	10 μs	100 μs
$n \log_2 n$	10 μs	133 μs	1.7 ms
n^2	1 ms	100 ms	10 s
n^3	1 s	16.7 min	11.6 days
n^4	16.7 min	116 days	3171 yr
2^n	$3.4 \cdot 10^{284}$ yr	$6.3 \cdot 10^{2993}$ yr	$3.2 \cdot 10^{30086}$ yr

Testing and Debugging

Program Testing

- Program testing occurs when you run a program and observe its behavior
- Part of the science of software engineering is the systematic construction of **a set of test inputs that is likely to discover errors**, and such test inputs are the topic of this section

Choosing Test Data

- To serve as good test data, your test inputs need two properties:
  1. You must know what output a correct program should produce for each test input
  2. The test inputs should include those inputs that are most likely to cause errors

Choosing Test Data (Cont'd)

Two methods for finding test data that is most likely to cause errors:

- **First Method:** Based on identifying and testing inputs called **boundary values**, which are particularly apt to cause errors
- A boundary value of a problem **is an input that is one step away from a different kind of behavior**

❑Example:

```
int time_check(int hour);
// Precondition: hour lies in the
// range 0 <= hour <= 23
```

- Two boundary values for `time_check` are hour equal to 0 and hour equal to 23

Choosing Test Data (Cont'd)

- If you cannot test all possible inputs, at least test the boundary values
- For example, if legal inputs range from 0 to 1000000, then be sure to test input 0 and input 1000000
- It is a good idea also to consider 0, 1, and –1 to be boundary values whenever they are legal input

Choosing Test Data (Cont'd)

Second Method: Fully exercising code requires intimate knowledge of how a program has been implemented

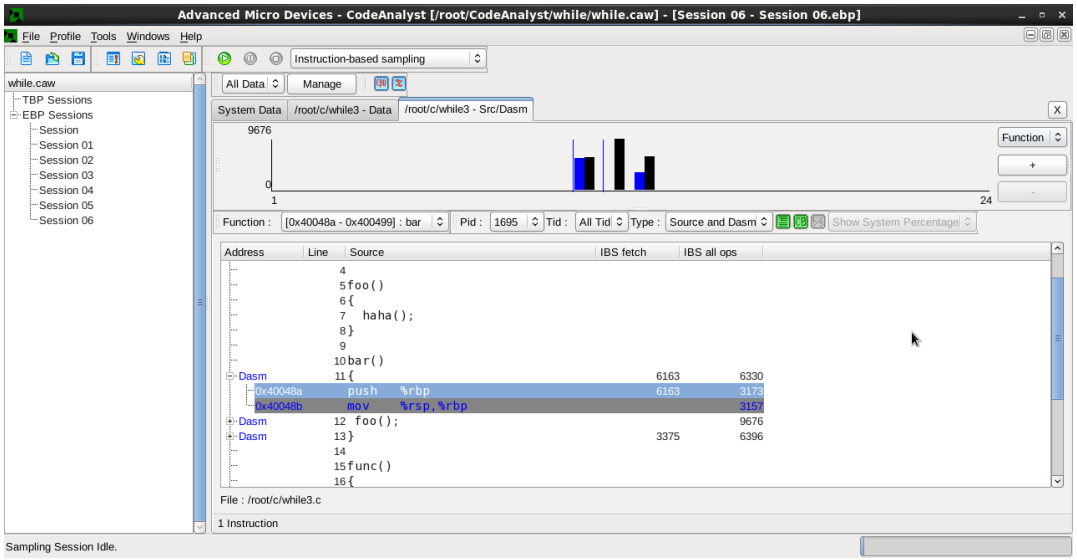
This technique is simple, with two rules:

- Make sure that **each line of your code is executed at least once** by some of your test data
- If there is some part of your code that is sometimes skipped altogether, then make sure that **there is at least one test input that actually does skip this part** of your code

Choosing Test Data (Cont'd)

- Many compilers have a software tool called a **profiler** to help fully exercise code
- A typical profiler will generate a listing indicating how often each statement of your program was executed
- This can help you spot parts of your program that were not tested

- Valgrind
- Oprofile
- Gprof
- AMD Code Analyst



Debugging (Cont'd)

- After an erroneous test input is found, you still must determine exactly why the "bug" occurs, and then "debug the program"
- Debugging Tip:
  1. Never start changing suspicious code on the hope that the change "might work better"
  2. Instead, discover exactly why a test case is failing **and limit your changes to corrections of known errors**
  3. Once you have corrected a known error, all test cases should be rerun

- Use a software tool called a **debugger** to help track down exactly why an error occurs

Summary

- One good method for specifying what a function is supposed to do is to provide a **precondition** and **postcondition** for the function
- Understanding and using the **C++ Standard Library** can make program development easier
- **Time analysis** is an analysis of how many **operations** an algorithm requires
- Three important examples of big-O analyses are **linear** ($O(n)$), **quadratic** ($O(n^2)$), and **logarithmic** ($O(\log n)$) .
- An important testing technique is to identify and test **boundary values**
- A second important testing technique is to ensure that your test cases are **fully exercising the code**

Appendix 1: **Time Complexity of "Guessing Game"**

Appendix 1: Time Complexity of Guessing Game

Guessing Game Function for the Time Analysis Example

```
#include <cassert>   // Provides assert
#include <iostream>  // Provides cout and cin
#include <cstdlib>   // Provides EXIT_SUCCESS
using namespace std; // Allows all Standard Library items to be used


// Prototype for the function used in this program
void guess_game(int n);
// Precondition: n > 0.
// Postcondition: The user has been asked to think of a number
// between 1 and n. The function asks a series of questions,
// until the number is found.


int main( )
{
    guess_game(100);
    return EXIT_SUCCESS;
}
```

Appendix 1: Time Complexity of Guessing Game

Guessing Game Function for the Time Analysis Example (Cont'd)

```
void guess_game(int n)
// Library facilities used: cassert, iostream
{
    int guess;
    char answer;

    assert(n >= 1);
```

One comparison, one call to assert = 2 operations

4 output operations

```
    cout << "Think of a whole number from 1 to " << n << "." << endl;
```

```
    answer = 'N';
```

One assignment operation

Appendix 1: Time Complexity of Guessing Game

Guessing Game Function for the Time Analysis Example (Cont'd)

Loop initialization = 1 operation

```
for (guess = n; (guess > 0) && (answer != 'Y') && (answer != 'y');
                                                    --guess)

{
    cout << "Is your number " << guess << "?" << endl;
    cout << "Please answer Y or N, and press return: ";
    cin >> answer;
}
```

***k* operations inside the loop**

Appendix 1: Time Complexity of Guessing Game

Guessing Game Function for the Time Analysis Example (Cont'd)

3 operation in the if statement

```
    if ((answer == 'Y') || (answer == 'y'))
```

2 operations

```
        cout << "I knew it all along." << endl;

    else
        cout << "I think you are cheating!" << endl;
}
```

References

1) Data Structures and Other Objects Using C++, Michael Main, Walter Savitch, 4th Edition
2) Data Structures and Algorithm Analysis in C++, by Mark A. Weiss, 4TH Edition
3) C++: Classes and Data Structures, by Jeffrey Childs
4) http://en.cppreference.com
5) http://www.cplusplus.com
6) https://isocpp.org