CSCI-1200 Computer Science II — Fall 2006
Lecture 2 — Algorithm Analysis & Strings

Announcements

- HW 1 is available on-line through the course website.
- If you have not resolved issues with the C++ environment on your laptop, please do so immediately.

Today

- Algorithm Analysis / Order Notation
- Scope
- Strings
- Loop Invariants

Readings: Koenig and Moo Chapters 1 & 2 and Malik pp 399-410

2.1 Algorithm Analysis

Why should we bother?

- We want to do better than just implementing and testing every idea we have.
- We want to know why one algorithm is better than another.
- We want to know the best we can do. (This is often quite hard.)

How do we do it? There are several options, including:

- Don't do any analysis; just use the first algorithm you can think of that works.
- Implement and time algorithms to choose the best.
- Analyze algorithms by counting operations while assigning different weights to different types of operations based on how long each takes.
- Analyze algorithms by assuming each operation requires the same amount of time. Count the total number of operations, and then multiply this count by the average cost of an operation.

2.2 Exercise: Counting Example

- Suppose `arr` is an array of `n` doubles. Here is a simple fragment of code to sum of the values in the vector:

```
double sum = 0;
for (int i=0; i<n; ++i)
  sum += arr[i];
```

- What is the total number of operations performed in executing this fragment? Come up with a function describing the number of operations *in terms of n*.
- We are likely to come up with different answers. How do we resolve these differences?

2.3 Which Algorithm is Best?

An venture capitalist is trying to decide which of 3 startup companies to invest in and has asked for your help. Here's the timing data for their prototype software on some different size test cases:

n	foo-a	foo-b	foo-c
10	10 u-sec	5 u-sec	1 u-sec
20	13 u-sec	10 u-sec	8 u-sec
30	15 u-sec	15 u-sec	27 u-sec
100	20 u-sec	50 u-sec	1000 u-sec
1000	?	?	?

Which company has the "best" algorithm?

2.4 Order Notation Definition

In CSII we will focus on the intuition of order notation. For more details and more technical depth, see any textbook on data structures and algorithms.

- Definition: Algorithm A is order $f(n)$ — denoted $O(f(n))$ — if constants k and n_0 exist such that A requires no more than $k * f(n)$ time units (operations) to solve a problem of size $n \geq n_0$.
- For example, algorithms requiring $3n+2$, $5n-3$, and $14+17n$ operations are all $O(n)$. This is because we can select values for k and n_0 such that the definition above holds. (What values?)
- Likewise, algorithms requiring $n^2/10 + 15n - 3$ and $10000 + 35n^2$ are all $O(n^2)$.
- Intuitively, we determine the order by finding the *asymptotically dominant term (function of n)* and throwing out the leading constant. This term could involve logarithmic or exponential functions of n.
- Implications for analysis:
  - We don't need to quibble about small differences in the numbers of operations.
  - We also do not need to worry about the different costs of different types of operations.
  - We don't produce an actual time. We just obtain a rough count of the number of operations. This count is used for comparison purposes.
- In practice, this makes analysis relatively simple, quick and (sometimes unfortunately) rough.

2.5 Common Orders of Magnitude

Here are the most commonly occurring orders of magnitude in algorithm analysis:

- $O(1)$, *a.k.a. CONSTANT*: The number of operations is independent of the size of the problem. e.g., compute quadratic root.
- $O(\log n)$, *a.k.a. LOGARITHMIC.* e.g., dictionary lookup, binary search.
- $O(n)$, *a.k.a. LINEAR.* e.g., sum up a list.
- $O(n \log n)$, e.g., sorting.
- $O(n^2)$, $O(n^3)$, $O(n^k)$, *a.k.a. POLYNOMIAL.* e.g., find closest pair of points.
- $O(2^n)$, $O(k^n)$, *a.k.a. EXPONENTIAL.* e.g., Fibonacci, playing chess.

2.6 Exercise: A Slightly Harder Example

- Here's an algorithm to determine if the value stored in variable `x` is also in an array called `foo`

```
int loc=0;
bool found = false;
while (!found && loc < n) {
  if (x == foo[loc])
    found = true;
  else
    loc ++ ;
}
if (found) cout << "It is there!\n";
```

- Can you analyze it? What did you do about the `if` statement? What did you assume about where the value stored in `x` occurs in the array (if at all)?

2.7 Best-Case, Average-Case and Worst-Case Analysis

- For a given fixed size vector, we might want to know:
  - The fewest number of operations (best case) that might occur.
  - The average number of operations (average case) that will occur.
  - The maximum number of operations (worst case) that can occur.
- The last is the most common. The first is rarely used.
- On the previous algorithm, the best case is $O(1)$, but the average case and worst case are both $O(n)$.

2.8 Approaching An Analysis Problem

- Decide the important variable (or variables) that determine the "size" of the problem.
  - For arrays and other "container classes" this will generally be the number of values stored.
- Decide what to count. The order notation helps us here.
  - If each loop iteration does a fixed (or bounded) amount of work, then we only need to count the number of loop iterations.
  - We might also count specific operations. For example, in the previous exercise, we could count the number of comparisons.
- Do the count and use order notation to describe the result.

2.9 Exercises: Order Notation

For each version below, give an order notation estimate of the number of operations as a function of `n`:

1.
```
int count=0;
for (int i=0; i<n; ++i)
  for (int j=0; j<n; ++j)
    ++count;
```

2.
```
int count=0;
for (int i=0; i<n; ++i)
  ++count;
for (int j=0; j<n; ++j)
  ++count;
```

3.
```
int count=0;
for (int i=0; i<n; ++i)
  for (int j=i; j<n; ++j)
    ++count;
```

2.10 Scope

- The *scope* of a name (identifier) is the part of the program in which it has meaning. Curly braces, `{ }`, establish a new scope — this includes functions and compound statements.
- Scopes may be nested.
- Identifiers may be re-used as long as they are in different scopes. Identifiers (variables or constants) within a scope hide identifiers within an outer scope having the same name. This does not change the values of hidden variables or constants — they are just not accessible.
- When a `}` is reached, a scope ends. All variables and constants (and other identifiers) declared in the scope are eliminated, and identifiers from an outer scope that were hidden become accessible again in code that follows the end of the scope.
- The operator `::` (namespaces) establishes a scope as well.

2.11 Scope Exercise

The following code will not compile. Why not? Fix it (minimally) & determine the output.

```
int main() {
  int a = 5, b = 10;
  int x = 15;
  {
    double a = 1.5;
    b = -2;
    int x = 20;
    int y = 25;
    cout << "a = " << a << ", b = " << b << endl << "x = " << x << ", y = " << y << endl;
  }
  cout << "a = " << a << ", b = " << b << endl << "x = " << x << ", y = " << y << endl;
  return 0;
}
```

2.12 String Example (from Koenig and Moo Chapter 1)

```
// ask for a person's name, and generate a framed greeting
#include <iostream>
#include <string>

int main() {
  std::cout << "Please enter your first name: ";
  std::string name;
  std::cin >> name;

  // build the message that we intend to write
  const std::string greeting = "Hello, " + name + "!";

  // build the second and fourth lines of the output
  const std::string spaces(greeting.size(), ' ');
  const std::string second = "* " + spaces + " *";

  // build the first and fifth lines of the output
  const std::string first(second.size(), '*');

  // write it all
  std::cout << std::endl;
  std::cout << first << std::endl;
  std::cout << second << std::endl;
  std::cout << "* " << greeting << " *" << std::endl;
  std::cout << second << std::endl;
  std::cout << first << std::endl;

  return 0;
}
```

2.13 About STL String Objects

- A `string` is an object type defined in the standard library to contain a sequence of characters.
- The string type, like all types (including `int`, `double`, `char`, `float`), defines an interface, which includes construction (initialization), operations, functions (methods), and even other types(!).
- When an object is created, a special function is run called a "constructor", whose job it is to initialize the object. The greeting example code exhibits three ways of constructing string objects:
  - By default to create an empty string
  - With a specified number of instances of a single char
  - From another string
- The notation `greeting.size()` is a call to a function `size` that is defined as a **member function** of the `string` class. There is an equivalent member function called `length`.
- Input to string objects through streams (e.g. reading from the keyboard or a file) includes the following steps:
  1. The computer inputs and discards white-space characters, one at a time, until a non-white-space character is found.
  2. A sequence of non-white-space characters is input and stored in the string. This overwrites anything that was already in the string.
  3. Reading stops either at the end of the input or upon reaching the next white-space character (without reading it in).
- The (overloaded) operator '+' is defined on strings. It concatenates two strings to create a third string, without changing either of the original two strings.
- The assignment operation '=' on strings overwrites the current contents of the string.

This seems like a lot to remember. Do I need to memorize this? Where can I find all the details on `string` *objects?*

2.14 Short Exercises

1. What will be the values of strings `a`, `b` and `c` at the end of the following code fragment:

```
std::string a, b, c;
std::cin >> a >> b >> c;
```

   for the input:

```
all-cows   eat123
      grass.  every good boy
deserves fudge!
```

2. Write a C++ code fragment that reads in two strings, outputs the shorter string on one line of output, and then outputs the two strings concatenated together with a space between them on the second line of output.

2.15 C++ vs. Java

- Standard C++ library `std::string` objects behave like a combination of Java `String` and `StringBuffer` objects. If you aren't sure of how a `std::string` member function (or operator) will behave, check its semantics or try it on small examples (or both, which is preferable).
- Java objects must be created using `new`, as in:

```
String name = new String("Chris");
```

  This is not necessary in C++. The C++ (approximate) equivalent to this example is:

```
std::string name("Chris");
```

  Note: There is a `new` operator in C++ and its behavior is somewhat similar to the new operation in Java. We will study it later in the semester.

2.16 More on Strings

- The individual characters of a string can be accessed using the subscript operator `[]` (similar to arrays).
  - Subscript 0 corresponds to the first character.
  - For example, given `std::string a = "Susan";`
    Then `a[0] == 'S'` and `a[1] == 'u'` and `a[4] == 'n'`.
- Strings define a special type `string::size_type`, which is the type returned by the string function `size()` (and `length()`).
  - The `::` notation means that `size_type` is defined within the scope of the `string` type.
  - `string::size_type` is generally equivalent to `unsigned int`.
  - You will have compiler warnings and potential compatibility problems if you compare an `int` variable to `a.size()`.

2.17 Example Problem: Writing a Name Along a Diagonal

We will spend the rest of lecture on the following problem: read in a name and then write it along a diagonal, framed by asterisks. Here's how the program should behave:

```
What is your first name? Bob

*******
*     *
* B   *
*  o  *
*   b *
*     *
*******
```

We will start by solving a simpler version and then look at two ways to solve the whole problem.

2.18 Exercise: Writing the Name Diagonally

Finish the program below to output the name diagonally, so that the program interaction looks like this:

```
What is your first name? Sally
S
 a
  l
   l
    y
```

Hint: You will need to use nested `for` *loops OR a single loop and exploit properties of the* `string` *class.*

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  cout << "What is your first name? ";
  string first;
  cin >> first;









  return 0;
}
```

2.19 Thinking About the Whole Problem

- Now we are better prepared to address the original problem. Here are the two main difficulties:
  - Making sure that we can put the characters in the right places on the right lines.
  - Getting the asterisks in the right positions and getting the right number of blanks on each line.
- In the previous example, we addressed the first point. With that practice, we are ready to try the full problem:
  - Initial stuff: read the name, and output a blank line, as in an earlier example.
  - Let's think about the main output: think of the output region as a grid of rows and columns: How big is this region? What gets output where?
  - This leads to an implementation with two nested loops, and conditionals used to guide where characters should be printed.

2.20 1st Solution: Grid of Characters

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  cout << "What is your first name? ";
  string first;
  cin >> first;

  const string star_line(first.size()+4, '*');
  const string blanks(first.size()+2, ' ');
  const string empty_line = '*' + blanks + '*';

  cout << endl << star_line << endl << empty_line << endl;
```

```
  // Output the interior of the framed greeting, one line at a time.
  for (string::size_type i = 0; i < first.size(); i++) {

    // Outputs the i-th row: *'s in the first (0-th) and last columns,
    // the i-th letter in column i+2, and a blank everywhere else.
    for (string::size_type j = 0; j < first.size()+4; ++ j) {
      if (j == 0 || j == first.size()+3)
        cout << '*';
      else if (j == i+2)
        cout << first[i];
      else
        cout << ' ';
    }
    cout << endl;
  }

  cout << empty_line << endl << star_line << endl;
  return 0;
}
```

2.21 Loop Invariants

- Definition: a *loop invariant* is a logical assertion that is true at the start of each iteration of a loop.
- An invariant can be stated in a comment, but it is not part of the actual code. It helps determine:
  - The conditions that may be assumed to be true at the start of each iteration.
  - What should happen in each iteration.
  - What must be done before the next iteration to restore the invariant.
- Analyzing the code relative to the stated invariant also helps explain the code and think about its correctness.

2.22 L-Values and R-Values

- Consider the simple code

  ```
  string a = "Kim";
  string b = "Tom";
  a[0] = b[0];
  ```

  String `a` is now `"Tim"`. No big deal, right? Wrong!

- Let's look closely at the line: `a[0] = b[0];` and think about what happens.

  In particular, what is the difference between the use of `a[0]` on the left hand side of the assignment statement and `b[0]` on the right hand side?

- Syntactically, they look the same. But,
  - The expression `b[0]` gets the char value, `'T'`, from string location 0 in `b`. This is an *r-value*.
  - The expression `a[0]` gets a reference to the memory location associated with string location 0 in `a`. This is an *l-value*.
  - The assignment operator stores the value in the referenced memory location.

  The difference between an *r-value* and an *l-value* will be especially significant when we get to writing our own operators later in the semester

- Has anyone seen the error message: "`non-lvalue in assignment`"? What's wrong with this code?

  ```
  std::string foo = "hello";
  foo[2] = 'X';
  cout << foo;
  'X' = foo[3];
  cout << foo;
  ```

2.23 Ideas for a 2nd Solution: Loop Invariant Practice

- Think about what changes from one line to the next.
- Suppose we had a "blank line" string, containing only the beginning and ending asterisks and the spaces between.
- We could overwrite the appropriate blank character, output the string, and then replace the blank character (and restoring the loop invariant).

2.24 Exercise: Finish the 2nd Solution

```
#include <iostream>
#include <string>

using std::cin;
using std::cout;
using std::endl;
using std::string;

int main() {
  cout << "What is your first name? ";
  string first;
  cin >> first;

  const string star_line(first.size()+4, '*');
  const string blanks(first.size()+2, ' ');
  const string empty_line = '*' + blanks + '*';
  string one_line = empty_line;

  cout << endl << star_line << endl << empty_line << endl;









  cout << empty_line << endl << star_line << endl;

  return 0;
}
```

Be sure to practice adding comments with your assumptions about the loop invariant.

2.25 Thinking About Problem Solving

- We began by working on simplified versions of the problem to get a "feel" for the core issues.
- Then we worked through two different solution approaches:
  - Thinking of the output as a two-dimensional grid and using logical operations to figure out what to output at each location.
  - Thinking of the output as a series of strings, one string per line, and then thinking about the differences between lines.
- There are often many ways to solve a programming problem. Sometimes you can think of several, while sometimes you struggle to come up with one.
- When you have finished a problem or when you are thinking about programming examples, it is useful to think about the core ideas used. If you can abstract and understand these ideas, you can later apply them to other problems.