ICS 353: Design and Analysis of Algorithms

Summer Term 2014 (2013-3)

Overview of Algorithms

Dr. Nasir Al-Darwish
Computer Science Department
King Fahd University of Petroleum and Minerals
darwish@kfupm.edu.sa

The Field of Algorithms

- The ***field of algorithms***, as a branch of computer science and modern mathematics, focuses on developing and formalizing problem-solving techniques.
- *Definition*: An *algorithm* is a step-by-step procedure for solving a problem by computer.
- The term "algorithm" is in reference to the Persian mathematician Al-Khwarizmi (780-850) — Al Khwarizmi's book *Hisab al-jabr w'al-muqabala* is the first known book on algebra.
- The concept of an algorithm originated as a means of recording procedures for solving mathematical problems such as *finding the common divisors of two integers* or *multiplying two numbers*.
- One of the oldest (and useful) algorithms, is *Euclid's GCD algorithm* for *finding the greatest common divisor of two integers*.
- The study and development of algorithms became a scientific endeavor with the invention of digital computers in the late 1940s and early 1950s, because computers can be programmed to process data and execute logical and arithmetic calculations at high speeds.

Algorithm Development: Pseudocode

- Today's computers are designed to compile and execute programs written in some high-level programming language such as C or Java.
- However, because algorithms are intended to be read and understood by humans, algorithms are better expressed in *pseudocode*.
- Pseudocode uses a mix of English, abstract mathematical notation and high-level programming language constructs (such as variable assignment and flow-control statements).
- Describing an algorithm in pseudocode frees the algorithm designer from worrying about the specific syntax of a particular programming language. Also, it is easier to analyze the algorithm and reason about its correctness when it is expressed using as simple notation as possible.
- Algorithm development is not an end of itself. It is just a means to efficiently solve a problem by computer.

Algorithm Development: Design vs. Analysis

- The study of algorithms evolves around two interrelated tasks: *design* and *analysis*.
- The *design of an algorithm* for a given problem simply means to develop a procedure (normally expressed in pseudocode) that manipulates the problem's input to obtain the desired output.
- This is followed up by the *algorithm analysis* task whose goal is *to quantify the algorithm's running time* (or *memory space usage*) as a function of input size.
- These tasks are interrelated because the results of the analysis often lead the algorithm designer to rethink and modify his algorithm.
- The final form of an algorithm is its embodiment in a computer program, which gives a materialistic assurance of its utility, efficiency, and correctness.
- Recognition of the importance of algorithms has led Donald Knuth to state, "Computer Science is the study of algorithms" [Knu74].
  - [Knu74] Knuth, D.E. Computer Science and Its Relation to Mathematics, *The American Mathematical Monthly*, vol. 81, No. 4 (Apr., 1974), pp. 323-343.

Means of Describing (Expressing) Algorithms

- The process of developing an algorithm for some given problem begins with characterizing the problem input and output, which represent *a level of abstraction* (approximate model) of reality.
- An algorithm is then a procedure (and eventually a computer program) that manipulates the problem's input in order to produce the desired output.
- The following various means are used to express an algorithm and convey it to its intended readers:
  - *Textual and pictorial description*: used to explain the idea and/or logic of the algorithm and any data structures needed for efficient implementation
  - *Pseudocode*: a standard way to express algorithms, Java-like code
  - *Mathematical equations*: a common and formal way to express an algorithm (or some of its parts)
  - *Program flowcharts* (outdated)

Describing Algorithms: Pseudocode

- Pseudocode uses a mix of English, mathematical notation and high-level programming language constructs (such as variable assignment and flow control statements).
- Algorithm statements are expressed using Pascal-like (or Java-like) syntax but with relaxed syntactic rules.
  - Use line breaks instead of semicolons to separate statements
  - Use variables without explicitly declaration of the variable's data-type
  - Use high-level operations (like the construct $A \leftarrow B$, to mean copy array B to array A) that have no corresponding constructs in typical programming languages.

Describing Algorithms: Pseudocode

- The following table shows examples of CSharp/Java program statements and the alternative forms used in pseudocode.

Csharp/Java Program Statement	Alternative Form Used in Pseudocode
`int a = 10;`	`a ← 10`
`if (a==b) S1 else S2;`	`if a=b then S1 else S2`

`for(int i=1; i <= n; i++)`
`{ S1; S2; }`
	`for i = 1 to n`
	`  S1`
	`  S2`
	`end for`

`while ((a != b) && (c <= d))`
`{ S1; S2; }`
	`while (a ≠ b) and (c ≤ d)`
	`  S1`
	`  S2`
	`end while`

Describing Algorithms: Mathematical Equations, Euclid's GCD

- Mathematical equations (functions) are handy in defining recursive algorithms — a definition is *recursive* if it exhibits a self-reference.
- A recursive definition (formulation) normally utilizes two equations: a *recursive equation* and a *base* (*nonrecursive*) *equation*.
- Normally, the base equation deals with special (or small) inputs for which the solution can be specified directly.
- *Example*: Euclid's GCD algorithm for computing the greatest common divisor (*gcd*) of two positive integers a, b where $a \geq b$. The algorithm is specified by these equations (***a mod b*** is the remainder of division of a by b):

$gcd(a,b) = b$ if a mod $b = 0$* (1) // Base Eq.

$gcd(a,b) = gcd(b, a \bmod b)$ if a mod $b > 0$ (2)

*This is same as if *b divides a* (or *a is divisible by b*)

- ***GCD-Example* (2 is GCD of 82 and 12):**

  $gcd(82,12)=gcd(12, 82 \bmod 12)=gcd(12,10)=gcd(10, 12 \bmod 10)=gcd(10,2)=2.$

- *The above lines simply corresponds to the (recursive) calls that takes place.* http://faculty.kfupm.edu.sa/ics/darwish/JS_AlgorithmAnimation/gcd_animated.htm

GCD in Pseudocode: Java/CSharp, JavaScript

- Translating an algorithm specified by recursive equations into pseudocode is often straightforward and purely mechanical.
- We merely rewrite the equations using a different syntax. For example, based on the previous equations, we can express the preceding GCD algorithm as follows :

```
int gcd(int a, int b) // GCD in Java/CSharp
{ // returns the GCD of the positive integers a and b (a > b)
  if (a % b == 0) return b;  // Base equation
  else return gcd(b, a % b); // Recursive equation
}
// GCD in JavaScript
function gcd(a, b)
{ // returns the GCD of the positive integers a and b (a > b)
  if (a % b == 0) return b;  // Base equation
  else return gcd(b, a % b); // Recursive equation
}
```

Euclid's GCD: Termination

- ***Important Note*:** To guarantee termination, the recursive equation must decrease the values of its input parameters to eventually reach the parameter-values handled by the base equation.

- For the GCD algorithm, observe that the first parameter is decreased from a to b, because we have assumed that $a \geq b$, and the second parameter is decreased from b to (a *mod* b), because division of a by b always leaves a remainder $< b$.

Algorithm-Design Guidelines

- We summarize certain guidelines (or blueprints) that govern the process of algorithm development. These guidelines translate into desirable characteristics that should be exhibited by the algorithm pseudocode-description.
  - *Modularity*
  - *Readability*
  - *Correctness*
  - *Time and space efficiency*
- These characteristics, together, ensure that the algorithm can be properly analyzed, make it easier to prove the algorithm correct, ease the manual process of translating the algorithm into a computer program, and, finally, ensure that the algorithm, timewise (and spacewise), is practical.

Algorithm-Design Guidelines: Modularity

- Modularity is an important principle in system design, where a large complex structure is constructed from smaller (simpler) building blocks.
- Using this principle in algorithm design, an algorithm is evolved by a process of *stepwise refinement*.
- This means that the initial description is composed of high-level steps. Then each of these high-level steps is expanded into simpler, more refined steps.
- The process is repeated until the description has enough details that it can be translated into a computer program in a non-ambiguous way using moderate effort. The process is generally known as the *top-down design principle*.
- Modularity calls for dividing a large monolithic description into a number of smaller-sized procedures, where each procedure embodies some specific functionality.
- **Note:** In the context of high-level programming languages, the term *module* is used as a synonym for a *class*, which defines some type of object and its behavior as a set of program functions.

Algorithm-Design Guidelines: Readability

- It is essential that the algorithm pseudocode-description be readable.
- Readability is achieved by
  - Using meaningful variable names,
  - Using indentation to indicate nesting of compound statements such as nested-loop structures, and
  - Using comments to annotate (and explain the actions of) various parts of the algorithm.

Algorithm-Design Guidelines: Correctness

- It is essential that an algorithm for a given problem produce the correct output for all possible inputs (problem instances).
- Often it is not feasible to ensure the algorithm correctness by testing the algorithm on an instance-by-instance basis, as there could be a large number of problem instances; rather, a mathematical proof of correctness must be given.
- As we will see later, algorithms developed by induction (a well-known algorithm-design technique) embody their own proof of correctness.
- Another algorithm proof method employs the concept of a loop invariant, which is an assertion about a for-loop (or a while-loop) block that remains valid in every iteration of the loop.

Algorithm-Design Guidelines: Time and space efficiency

- Computer processor time (i.e., time spent by the CPU executing a program) and computer memory are considered precious and scarce resources that ought to be used efficiently.
- This is because in reality a computer is always shared among many competing tasks and is rarely dedicated to executing a single task.
- Often, *computer time is viewed as a more precious resource than computer memory* because computer time is human time, as there is always a person waiting for a running program to finish.
- Therefore, for solving a given problem, we always seek a fast algorithm that uses a reasonable amount of computer memory.

Can we use JavaScript for coding algorithms?

- JavaScript is a "scripting" programming language.
- JavaScript execution uses "interpreted" code, instead of "compiled" code. Thus, it is slower in comparison with Java or CSharp.
- It is limited in features for security reasons; for example, JavaScript cannot access files on the user's hard-disk.
- The really nice thing about JavaScript is that it runs within a web browsers
  - Fully supported by all modern browsers
  - The program interface can be easily built using HTML
  - HTML DOM provides a standardized way of manipulating HTML
- JavaScript has proven to be an "enabling technology". It allows for quick development and experimentation of "awesome" ideas.
  - Algorithm Animation: http://faculty.kfupm.edu.sa/ics/darwish/ICS353-Summer2014/links.htm
  - D3 (Data Driven Documents): http://bl.ocks.org/mbostock/3750558

Sequential Search in JavaScript

```
<script type = "text/javascript"> // Place script within Head section
function ExecuteSearch()
{   var A = [4,8,13,17,19,22,25,35,39,41,49,53,59,67,69,75,78,83,90,99];
    var searchKey = inputVal.value;
    alert("input key=" + searchKey);
    console.log("input key=" + searchKey);
    // searchKey = parseInt(searchKey);  // not needed
    var position = Search(A, searchKey);
    if (position != -1)
       result.value = "The number is found at position " + position;
    else result.value = "The number is not found";
}
// Search array A for the specified "key" value
function Search(A, key)
{  for (var i = 0; i < A.length; i++)
   { if (A[i] == key) return i; }
   return -1;
}
</script>
// ...  Some HTML goes here
<p>Enter an integer between 0 and 100
<br/><input id="inputVal" type="text" />
<input type="button" value="Search"  onclick="ExecuteSearch()" /><br/>
<p>Result <br/><input id= "result" type="text" size="40" />
```

Prerequisites for Effective use of JavaScript

- Some good books/links on JavaScript can be found on at http://faculty.kfupm.edu.sa/ics/darwish/SWE363-Fall2013/
- Know HTML DOM and Event Handling. A good book is *DOM Enlightenment* (http://domenlightenment.com/)
- Know how to debug scripts using the Brower's Developer Tools (Hit F12 or choose *Inspect Element* from context menu). For Google's Chrome, https://developer.chrome.com/devtools/docs/javascript-debugging
- You can develop and publish JS code at places like: jsfiddle, codepen, liveweave, jsbin
- Other JavaScript Issues
  - The division operator always produces a fractional number even if operands are integers (use Math.floor() to truncate the result)
  - Variables inside the body of a function not declared with "var" are treated as global (they exist after the function returns).
  - Always declare program variables using "var". The lack of it can cause bugs, especially for recursive functions.
  - To force explicit declaration, add the line `"use strict";` to the start of your script.