COP 3530 : Computer Science III
Design and Analysis of Data Structure and Algorithms

Course Particulars

Instructor: Sumanta Pattanaik
e-mail: sumant@cs.ucf.edu
ph: 823-2638
Meeting time: MW from 1:30PM to 2:45PM. At BA Rm# 119.
Office Hours: M-Th from 5:30PM to 6:30PM. At CSB Rm #251.

Course Particulars (continued)

Lab: M 3:30 PM to 4:20 PM in CSB #221
     M 4:30 PM to 5:20 PM in BA #115
     W 12:30 PM to 1:20 PM in BA #126
Lab Instructors:
    Weifeng Sun: CSB #113. Ph: 407-823-3934
    Hua Zhang: CC1 #203. Ph: 407-862-0154
Course Particulars (continued)

Text Book: Algorithm Design
by Michael Goodrich and Roberto Tomassia

Reference Books:
Data Structures & Algorithm Analysis in Java by Mark Weiss
Introduction to Algorithms by Cormen et al.

Resources:
Text's web site: http://algorithmdesign.net/

Course Particulars (continued)

Course Grading:

- Lab Quiz: 15% total
  - About 1 quiz every week.
- Programming Project: 30% total
  - 1 project early in the semester
  - 1 project towards the middle of the semester
  - 1 project towards the end of the semester
- Exam: 55% total
  - 2 Exams: 1 hr 15 min duration. 15% each.
  - Dates: Monday, September 29 and Wednesday, October 29.
  - 1 Final Exam: 2 hr 50 min duration. 25%.
  - Date: 1:00 PM-3:50PM, Monday December 8.

Course Particulars (continued)

Grades reported to the registrar's office will be letter grades
with +/-.
A: 90–100, A−: 88–90;
B+: 86–88, B: 80–86, B−: 78–80;
C+: 76–78, C: 70–78, C−: 68–70;
C: 66–68, C−: 60–68, D−: 58–60;
F: below 58.

Letter Grade to Grade Point conversion:
A = 4.00, A− = 3.75
B+: 3.50, B = 3.00, B− = 2.75
C+: 2.25, C = 2.00, C− = 1.75
D+: 1.25, D = 1.00, D− = 0.75
D = 1.00, F = 0.00
Course Particulars (continued)

**Course Policy:**
- UCF policy on academic honesty will be followed in the class.
- Students are responsible for keeping track of the material presented in class. Textbook, notes (if any) may not be sufficient to pass the quizzes and exams.
- All tests are closed book.
- No collaboration is allowed on test.
- Exam dates are as specified in the web page.
- There will be no retake of any of the exams.

Course Particulars (continued)

**Project Submission Guideline:**
- The projects that you submit must represent your own work and not that of a fellow student, past or present. You may never directly copy another student's code. Violating these guidelines may result in no credit for an assignment.
- Java is the programming language for the assignments.
- Assignments are due on the specified date. No credit for late submission.
- Complete project, (with source code, executable, write-up and output) must be submitted on or before the deadline.
- Assignment submission is through WebCT. http://webct.ucf.edu

Prerequisites of the course

**What will not be covered?**

- Elementary data structure, e.g. arrays, linked lists (Chapter 2).
- Object Oriented Programming, ADTs
- Java Programming.
Goals of this Course

What will be covered?
1. Analysis of the Algorithm.
3. New Data Structures.

Topics of the Course

- Algorithm Analysis: Chapter 1
- Search Trees and Skip Lists: Chapter 3
- Sorting, Sets and Selection: Chapter 4
- Fundamental Techniques: Chapter 5
- Graphs: Chapter 6
- Weighted Graphs: Chapter 7
- Network Flow and Matching: Chapter 8
- Text Processing: Chapter 9
- Computational Geometry: Chapter 12
- NP-Completeness: Chapter 13

Algorithms and Programs

Algorithm: a method to solve a problem.
- A recipe.

An algorithm takes the input to a problem and transforms it to the output.
- A mapping of input to output.

A problem can have many algorithms.
Algorithm Properties

An algorithm possesses the following properties:
- It must be composed of a finite number of steps.
- There can be no ambiguity as to which step will be performed next.
- It must terminate.
- It must be correct.

A computer program is an instance, or concrete representation, for an algorithm in some programming language.

Why Analyze?

There are often many approaches (algorithms) to solve a problem.

How do we choose between them?
1. an algorithm that is easy to understand, code, debug.
2. an algorithm that makes efficient use of the computer’s resources.

Efficiency

A solution is said to be efficient if it solves the problem within its resource constraints.
- Space
- Time

- Analyzing the solution is finding the amount of resources that the solution consumes.
Factors affecting resource use

For most algorithms, resource use depends on “size” of the input.

ex: Running time is expressed as $T(n)$ for some function $T$ on input size $n$.

![Diagram of input, algorithm, and output]

How to Measure Efficiency?

- Empirical study (run programs)
- Asymptotic Algorithm Analysis

Experimental Study (§ 1.6)

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like `System.currentTimeMillis()` to get an accurate measure of the actual running time
- Plot the results
How to Measure Efficiency?

- Empirical study (run programs)
- Asymptotic Algorithm Analysis

Tools: Pseudo Code

Notation for describing algorithms
- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Hides program design issues

Example: find max element of an array

Algorithm arrayMax(A, n)
Input array A of n integers
Output maximum element of A

currentMax ← A[0]
for i ← 1 to n − 1 do
  if A[i] > currentMax then
    currentMax ← A[i]
return currentMax

Pseudocode Details

- Method declaration
  Algorithm method(arg[, arg...])
  Input ...
  Output ...
- Control flow
  - if ... then ... [else ...]
  - while ... do ...
  - repeat ... until ...
  - for ... do ...
  - Indentation replaces braces

Example: find max element of an array

Algorithm arrayMax(A, n)
Input array A of n integers
Output maximum element of A

currentMax ← A[0]
for i ← 1 to n − 1 do
  if A[i] > currentMax then
    currentMax ← A[i]
return currentMax
Pseudocode Details

- Expressions
  - Assignment (like = in Java)
  - Equality testing (like == in Java)
  - Superscripts and other mathematical formatting allowed
- Return value
- Method call
  \( \text{var.method(arg[, arg…])} \)

Example: find max element of an array

Algorithm \( \text{arrayMax}(A, n) \)

Input array \( A \) of \( n \) integers
Output maximum element of \( A \)

\( \text{currentMax} \leftarrow A[0] \)
for \( i \leftarrow 1 \) to \( n-1 \) do
  if \( A[i] > \text{currentMax} \) then
    \( \text{currentMax} \leftarrow A[i] \)
return \( \text{currentMax} \)

Tools: The Random Access Machine (RAM) Model

- A single CPU
- Serial Computation
- An potentially unbounded bank of memory cells, each of which can hold an arbitrary number or character
- Memory cells are numbered and accessing any cell in memory takes unit time.

RAM Model: Primitive Operations

- Basic computations assumed to take a constant amount of time in the RAM model.
- Identifiable in pseudocode
- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method
Running Time Examples (1)

Example 1:
\[ a \leftarrow b \]
This assignment takes one unit of time.

Example 2:
\[ \text{sum} \leftarrow 0 \]
\[ \text{for } i \leftarrow 0 \text{ to } n-1 \text{ do } \]
\[ \text{sum} \leftarrow \text{sum} + n \]

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

**Algorithm arrayMax(A, n)**

\[
\begin{align*}
\text{currentMax} & \leftarrow A[0] \\
\text{for } i & \leftarrow 1 \text{ to } n-1 \text{ do } \\
& \quad \text{if } A[i] > \text{currentMax} \text{ then } \\
& \quad \quad \text{currentMax} \leftarrow A[i] \\
& \quad \quad \text{( increment counter } i ) \\
& \text{return currentMax}
\end{align*}
\]

Best, Worst, Average Cases

Not all inputs of a given size take the same time to run.

Sequential search for \( \text{ArrayMax} \) in an array of \( n \) integers:
- Begin at first element in array and look at each element in turn until \( \text{ArrayMax} \) is found

Best case:
Worst case:
Average case:
Which Analysis to Use?

While average time appears to be the fairest measure, it may be difficult to determine.

Worst case time analysis provides us with a robust major of the efficiency of the algorithm.

If statement: Take greater complexity of then/else clauses.

Switch statement: Take complexity of most expensive case.

Running Time Examples (2)

Example 3:

```plaintext
sum ← 0
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        sum ← sum + 1
```

Running Time Examples (3)

Example 4:

```plaintext
sum2 ← 0
for i ← 0 to n-1 do
    for j ← 0 to i do
        sum2 ← sum2 + 1
```
Tools: Math Fundamentals

- Summations:
  - Notation: \( \sum f(i) \)
  \[ \sum f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b) \]
  - Ex: Arithmetic Summation:
    \[ \sum_{i=1}^{n} = 1 + 2 + 3 + \cdots + n \]
    \[ \sum_{i=1}^{n} = \frac{n(n+1)}{2} \]

Math Fundamentals (continued)

- Summations:
  - Ex: Arithmetic Summation:
  - Proof:
    \[ S = 1 + 2 + 3 + \cdots + n \]
    \[ S = n + (n - 1) + (n - 2) + \cdots + 1 \]
    \[ 2S = (n + 1) + (n + 1) + (n + 1) + \cdots + (n + 1) \]
    \[ = n(n + 1) \]
    \[ S = \frac{n(n + 1)}{2} \]

Math Fundamentals (continued)

- Geometric Summation:
  - Ex:
    \[ \sum_{i=0}^{n} x^i = 1 + x + x^2 + \cdots + x^n \]
    \[ = \frac{1 - x^{n+1}}{1 - x} \text{ where } n \geq 0 \text{ and } x > 0 \text{ and } x \neq 1 \]
Math Fundamentals (continued)

- Summations:
  - ex: Geometric Summation
  - Proof:
    \[
    S = x + x^2 + \cdots + x^n = \frac{1 - x^{n+1}}{1 - x}
    \]
    \[
    S(1-x) = 1 - x^{n+1} \implies S = \frac{1 - x^{n+1}}{1-x}
    \]
    \[
    \sum_{i=1}^{n} x^i = \frac{1 - x^{n+1}}{1-x}
    \]
    
    if \(0 < x < 1\) then \(\sum_{i=1}^{n} x^i \leq \frac{1}{1-x}\)

- Summations:
  - \(\sum_{r=1}^{n} r^2 = \frac{n(n+1)(2n+1)}{6}\)
  - \(\sum_{r=1}^{n} r^3 = \frac{n^2}{3}\)
Math Fundamentals (continued)

- Summations:
  - Telescoping sum \( \sum_{n} (f(n) - f(n-1)) = f(n) - f(0) \)
  - Splitting a sum \( \sum_{n} f(n) = \sum_{n} f(n) + \sum_{n} f(n) \)

Running Time Examples (3)

Example 4:

```plaintext
sum2 ← 0
for i ← 0 to n-1 do
    for j ← 0 to i do
        sum2 ← sum2 +1
```

Other Control Statements

while loop: Analyze like a for loop.
Method call: Complexity of the method.
Computing Prefix Averages

- The $i$-th prefix average of an array $X$ is average of the first $(i+1)$ elements of $X$:

$$A[i] = \frac{X[0] + X[1] + \ldots + X[i]}{i+1}$$

### Prefix Averages (Method 1)

- prefix averages by applying the definition

#### Algorithm prefixAverages1($X$, $n$)

- **Input** array $X$ of $n$ integers
- **Output** array $A$ of prefix averages of $X$
- $A \leftarrow$ new array of $n$ integers
- for $i \leftarrow 0$ to $n-1$ do
  - $s \leftarrow X[0]$
  - for $j \leftarrow 1$ to $i$ do
    - $s \leftarrow s + X[j]$
  - $A[i] \leftarrow s / (i+1)$
- return $A$

### Prefix Averages (Method 1)

- The following algorithm computes prefix averages in quadratic time by applying the definition

#### Algorithm prefixAverages1($X$, $n$)

- **Input** array $X$ of $n$ integers
- **Output** array $A$ of prefix averages of $X$
- $A \leftarrow$ new array of $n$ integers
- for $i \leftarrow 0$ to $n-1$ do
  - $s \leftarrow X[0]$
  - for $j \leftarrow 1$ to $i$ do
    - $s \leftarrow s + X[j]$
  - $A[i] \leftarrow s / (i+1)$
- return $A$
Prefix Averages (Method 1)

• \texttt{prefixAverages1} runs in \((c_1 n^2 + c_2 n)\) time.

Prefix Averages (Method 2)

• The following algorithm computes prefix averages by keeping a running sum

\begin{algorithm}
\textbf{Algorithm} \texttt{prefixAverages2}(X, n)
\begin{itemize}
\item Input: array \(X\) of \(n\) integers
\item Output: array \(A\) of prefix averages of \(X\)
\end{itemize}
\begin{itemize}
\item \(A \leftarrow \) new array of \(n\) integers
\item \(s \leftarrow 0\)
\item \textbf{for} \(i \leftarrow 0\) to \(n-1\) \textbf{do}
\item \(s \leftarrow s + X[i]\)
\item \(A[i] \leftarrow s / (i + 1)\)
\end{itemize}
\textbf{return} \(A\)
\end{algorithm}

Prefix Averages (Method 2)

• The following algorithm computes prefix averages by keeping a running sum

\begin{algorithm}
\textbf{Algorithm} \texttt{prefixAverages2}(X, n)
\begin{itemize}
\item Input: array \(X\) of \(n\) integers
\item Output: array \(A\) of prefix averages of \(X\)
\end{itemize}
\begin{itemize}
\item \(A \leftarrow \) new array of \(n\) integers
\item \(s \leftarrow 0\)
\item \textbf{for} \(i \leftarrow 0\) to \(n-1\) \textbf{do}
\item \(s \leftarrow s + X[i]\)
\item \(A[i] \leftarrow s / (i + 1)\)
\end{itemize}
\textbf{return} \(A\)
\end{algorithm}

• Algorithm \texttt{prefixAverages2} runs in \(4n + 2\) time
Prefix Averages (Method 2)

- $\text{prefixAverages2}$ runs in $(c_3 n + c_4)$ time.

Efficiency

- Algorithm 1: runs in $(c_1 n^2 + c_2 n)$ time.

- Algorithm 2: runs in $(c_3 n + c_4)$ time.