CS 6212 DESIGN AND ANALYSIS OF ALGORITHMS

THIS LECTURE: INTRODUCTION

Instructor: Abdou Youssef

OBJECTIVES OF THIS LECTURE

By the end of this lecture, you will be able to:

- Describe basic algorithmic definitions
- Explain and begin to write pseudo code (pseudo-language)
- Differentiate between functions and procedures
- Show the <u>structure</u> of <u>valid</u> <u>recursion</u>
- Explain what analysis of algorithms means, and begin to utilize related elements, including
    - The big-O notation
    - The Master Theorem
    - Stirling's Approximation
    - Valuable summation formulas widely used in analysis of algorithms

OUTLINE

- Definitions and characteristics of algorithms, functions and procedures
- What is design of algorithms?
- What is analysis of algorithms?
- Pseudo language for expression algorithms
- Recursion
- Asymptotics (Big-O, Big-Ω, and Big-Θ notations): definitions
- Rules, theorems and formulas to help you derive Big-O for time complexity functions

PRELIMINARIES

- Purpose of the course:
  - Learn the design and analysis of algorithms

- Each keyword will be defined next
  - Algorithm
  - Design
  - Analysis

DEFINITION AND CHARACTERISTICS OF "ALGORITHM"

- Definition of Algorithm
  - A precise statement to solve a problem on a computer
  - A sequence of definite instructions to do a certain job
- Characteristics of Algorithms and Operations
  - Definiteness of each operation (i.e., clarity, unambiguity, single meaning)
  - Effectiveness (i.e., doability on a computer)
  - Termination in a finite amount of time
  - An algorithm has zero or more input, one or more output
- Special forms of algorithms:
  - Functions and procedures

Contrast with a program:
Does a program have to terminate?

Example of algorithm that takes 0 input?

More on that later

DESIGN
OF ALGORITHMS

- To design (an algorithm) for a problem means to:
  - Devise a method, using potentially a standard design technique, for solving the problem
  - Express the design (in a pseudo language, flowchart, etc.)
  - Validate the design/algorithm: Proof of correctness

- Note:
  - Implementation of algorithm often means coding it in a high-level programming language like C/C++, java, Python, etc.
  - Bulk of the course: learning standard algorithm-design techniques

DESIGN TECHNIQUES
THAT WILL BE COVERED IN THIS COURSE

- Divide and conquer
- The greedy method
- Dynamic programming
- Graph traversal
- Backtracking
- Branch and bound

ANALYSIS
OF ALGORITHMS

- What does "analysis" mean in general? (not just in the context of algorithms)
- Analysis of an algorithm means:
  - Determination time and space (i.e., memory) requirements of the algorithm
- Since memory has become abundant and cheap,
  - Analysis of algorithms is often reduced to just time analysis
- Fancy terms:
  - Time *complexity* analysis: determine the time requirements of an algorithm
  - Space *complexity* analysis: determine the space/memory requirements of an algorithm
- Complexity analysis is usually expressed in Big-O
  - giving only rough estimates
  - Caring more about asymptotic growth (i.e., growth trend as input size grows large)

More on Big-O later

EXPRESSION OF ALGORITHMS
-- PSEUDO LANGUAGE SYNTAX --

- Notes:
  - Words in **bold** are reserved words
  - We don't care if instructions are ended with semicolons (;) but it is preferred
- Variable declaration:
  - **integer** x, y; or **int** x, y;
  - **real** x, y; or **float** x, y; or **double** x,y;
  - **boolean** a , b; or **bool** a, b;
  - **character** z; or **char** z;
  - **string** s;
  - **generic x;** // if don't know/care about specific type, or if code works on several types
  - **Arrays:** **int** A[1:n], B[4:10]; **char** C[1:n]; and the like.
- Operations: +, -, *, /, ++, --, % (or mod or modulo), **and** (&, &&), **or** (|, ||), **not**
- Relations: <, <=, >, >=, ==, != (or ≠)

PSEUDO LANGUAGE
- ASSIGNMENTS -

- Assignments:
  - X = Expression;   or   X := Expression;   or   X ← Expression;
  - X **op=** Expr; // means: X = X **op** Expr;   **op** can be: +, -, *, /

- Examples:
  - x = 1+3*4;
  - y=2*x-5;
  - z= z+1;
  - x += 5;   // same as: x=x+5;
  - x=funct(a,b,c); // function call to a function "funct". More on that later

PSEUDO LANGUAGE
- CONTROL STRUCTURES: CONDITIONS -

<table>
<tr>
<td>
if condition then
        a sequence of statements;
[else
        a sequence of statements;]
endif

Note: Things between brackets […] are optional
</td>
<td>
if condition1 then
        a sequence of statements;
elseif condition2 then
        a sequence of statements;
    ……
elseif condition k then
        a sequence of statements;
[else
        a sequence of statements;]
endif
</td>
</tr>
<tr>
<td>
case x:
    Value1: statements; [break;]
    …
    Valuek: statements; [break;]
endcase
</td>
<td>
case:
    Cond1: statements; [break;]
    …
    Condk: statements; [break;]
    [Default: statements; [break;]]
endcase
</td>
</tr>
</table>

PSEUDO LANGUAGE
- CONTROL STRUCTURES: LOOPS -

while *condition* **do**
a sequence of statements;
endwhile
	loop
	a sequence of statements;
	until ***condition;***

for i= m **to** n **do**
a sequence of statements;
endfor
	for i= m **to** n **[step** d] **do**
	a sequence of statements;
	endfor

PSEUDO LANGUAGE
-- INPUT-OUTPUT --

- Input
  - **read**(X);   // X is a variable or array or even an elaborate structure. We'll rarely use it
- Output
  - **print**(data);
  - **write**(data, file);   // data can numeric or strings

PSEUDO LANGUAGE
-- FUNCTIONS AND PROCEDURES: DEFINITION --

- **Function**: An algorithm that
  - takes zero or more input parameters,
  - returns explicitly one output to the calling algorithm, and
  - can be called by other algorithms, which must provide the input parameters
- **Procedure**: An algorithms that
  - takes zero or more input parameters, and
  - computes one or more outputs by
    - writing into global variables (aka, side effect), and/or
    - Storing output in output parameters.
  - can be called by other algorithms, which must provide the input and output parameters
- More details and examples will follow after we see the pseudo-language

PSEUDO LANGUAGE
-- FUNCTIONS AND PROCEDURES: SYNTAX --

function *name*(*parameters*) // ok if you use "**func**" instead of "**function**"
begin // Ok to enclose code with braces {…} instead of **begin** … **end**
    variable declarations;
    sequence of statements;
    return (value);
end *name*

procedure *name*(**input** *params;* **output** *params;* **in-out** *params*)
begin
// Ok if you use "**proc**" instead of "**procedure**"
// Ok to enclose code with braces {…} instead of **begin** … **end**
    variable declarations;
    sequence of statements;
end *name*

PSEUDO LANGUAGE

-- FUNCTIONS AND PROCEDURES: EXAMPLES --

```
function max(A[1:n])
begin
   generic x=A[1];  // max so far
   int i;
   for i=2 to n do
      if (x<A[i]) then
         x=A[i];
      endif
   endfor
   return (x);    // observe the "return"
end max
```

```
procedure max(input A[1:n]; output M)
begin
   int i;
   M=A[1];
   for i=2 to n do
      if (M<A[i]) then
         M=A[i];
      endif
   endfor          // output is stored in M
end max    // observe: no "return" statement
```

PSEUDO LANGUAGE
-- ANOTHER PROC. EX.: IN-OUT PARAMETERS --

Procedure *swap*(**in-out x,y**) // swaps the values of x and y in place

Begin

  generic temp;

  temp=x;

  x=y;

  y=temp;

end swap

RECURSION
-- DEFINITION AND STRUCTURE --

- **Definition**: A *recursive algorithm* is an algorithm that calls itself on "smaller" input (smaller in size or value(s) or both).
- **Structure of recursive algorithms**:

Algorithm *name*(input)

begin

basis step; // for when the input is the smallest (in size/value); no calls to *name*(…).

name (smaller input) // A recursive call: the same algorithm *name* appears in the body

// there can be more statements and more recursive calls here

// the result of each recursive call is called a *subsolution*

Combine subsolutions; // or process subsolution(s) further

end

RECURSION
-- EXAMPLE--

```
function max(input A[i:j])          // finds the max of A[i],A[i+1],A[i+2],… ,A[j]
begin
        generic x, y;
        if (i==j) then      //input size is 1, which is the smallest
                return A[i];
        endif
        int m=(i+j)/2;
        x=max(A[i,m]);              // recursive call returning max of 1st half of the array
        y=max(A[m+1,j]);            // recursive call returning max of 2nd half of the array
        //next, merge the two sub-solutions into a global solution
        if (x<y) then
                return y;
        else
                return x;
        endif
end max
```

VALIDATION OF ALGORITHMS

- Validation: Proof of Correctness

- Often through proof by induction on the input size, such as in:
  - Recursive algorithms
  - Divide and conquer algorithms
  - Greedy algorithms
  - Dynamic programming algorithms
  - Sometimes when proving optimality of solutions

- Also, deductive methods of proofs.

ANALYSIS OF ALGORITHMS
-- DEFINITION AND PURPOSE --

- What: estimation of time and space (memory) requirements of the algorithm
- Reason/Purpose
  - Early estimation of performance to see if the algorithm meets speed requirements <u>before</u> any further investment of effort into the algorithm (i.e., before implementation))
  - If the algorithm is not fast enough, the designer must come up with <u>faster</u> algorithms
  - Complexity analysis is a way for comparing algorithms.:
    - one (or several competing designers) may design alternative algorithms for the same problem
    - you need to determine which to choose.
    - typically the fastest algorithm (and/or least demanding in memory) is chosen.

ANALYSIS OF ALGORITHMS
-- BASIC HOW-TO --

- Time complexity $T(n)$:
  - Number of operations in the algorithm, as a function of the input size.
  - Random access memory (RAM) model: Each of the arithmetic/Boolean operations and each of the relations (e.g., comparisons) are counted as one operation
- Space complexity $S(n)$: number of memory words needed by the algorithm
- Example:

function *max*(A[1:n])
begin
    generic x=A[1]; // max so far
    for i=2 **to** n **do**
        if (x<A[i]) **then**
            x=A[i];
        endif
    endfor
    return (x);
end max
	Time analysis:
	• n-1 comparisons
	• n assignments
	• Thus: T(n)=(n-1)+n=2n-1
	
	Space analysis:
	• Only one variable declared: x
	• Thus: S(n)=1
	• If take input size into account: S(n)=n+1

ANALYSIS OF ALGORITHMS
-- ART OF ASYMPTOTIC ESTIMATION --

- Since analysis is to determine if alg is fast enough or to choose b/w competing algs:
  - the time analysis need not be very accurate (down to the exact number of operations)
  - rather, an approximation of time is sufficient, and is often more convenient to derive
- Also, since speed usually matters more when input size n is large
  - we are more concerned about the "order of growth" of the time function $T(n)$, or as typically called, the *asymptotic behavior* of the $T(n)$.
- Since computers vary in speed from model to model and from generation to generation, and the variation is by a constant factor (with respect to input size):
  - we can (and should) ignore constant factors in time estimations, and
  - focus on the order of growth rather than the precise time in micro/nano-seconds.
- Therefore, a notation for approximation, for being "carefully careless", is needed: Big-O

ASYMPTOTICS AND BIG-O NOTATION

- **Definition**: let $f(n)$ and $g(n)$ be two functions of n (n is usually the input size). We say

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

if $\exists$ an integer n_0 and a positive constant k such that $|f(n)| \leq k|g(n)| \quad \forall n \geq n_0$.

$\exists$: there exists
$\forall$: for every

- **Examples**:
  - $3n + 1 = O(n^2)$ since $3n + 1 \leq 3n^2 \;\; \forall n \geq 2.\, n_0 = 2, k = 3.$

$$f(n) = 3n + 1, g(n) = n^2, |f(n)| \leq 3|g(n)| \; \forall n \geq 2$$

  - $3n + 6 = O(n)$ because $3n + 6 \leq 4n \;\; \forall n \geq 6.\, n_0 = 6, k = 4.$

$$f(n) = 3n + 6, g(n) = n, |f(n)| \leq 4|g(n)| \; \forall n \geq 6$$

ASYMPTOTICS AND BIG-Ω NOTATION

- **Definition**: let $f(n)$ and $g(n)$ as above. We say that

$$f(n) = \Omega(g(n))$$

if $\exists$ an integer n_0 and a positive constant k such that $|f(n)| \geq k|g(n)| \quad \forall n \geq n_0$.

- **Examples**:
  - $\frac{1}{3}n^2 = \Omega(n)$ because $\frac{1}{3}n^2 \geq n \quad \forall n \geq 3$. $n_0 = 3, k = 1$.

$$\boxed{f(n) = \frac{1}{3}n^2, g(n) = n, |f(n)| \geq |g(n)| \; \forall n \geq 3}$$

  - $3n + 6 = \Omega(n)$ because $3n + 6 \geq 3n \quad \forall n \geq 1$. $n_0 = 1, k = 3$.

$$\boxed{f(n) = 3n + 6, g(n) = n, |f(n)| \geq 3|g(n)| \; \forall n \geq 1}$$

ASYMPTOTICS AND BIG-Θ NOTATION

- **Definition**: let $f(n)$ and $g(n)$ as above. We say that

$$f(n) = \Theta(g(n))$$

if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. That is,

if $\exists$ an integer n_0 and two positive constant k_1 and k_2 such that

$$k_1|g(n)| \leq |f(n)| \leq k_2|g(n)| \quad \forall n \geq n_0.$$

- **Example**: $3n + 6 = \Theta(n)$ because $3n + 6 = O(n)$ and $3n + 6 = \Omega(n)$ as seen above

ASYMPTOTICS NOTES
-- USE OF BIG-O, Ω AND Θ --

- For most of the semester, we will be using the Big-O notation, but not much the Big-Ω or the Big-Θ

- The Big-Ω or the Big-Θ will be used in Lower Bound Theory near the end of the semester

BIG-O NOTES

- **Observation (Transitivity of the Big-O)**:
  - If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
  - Proof: An exercise
- Example:
  - Take $f(n) = 3n + 4, g(n) = n, h(n) = n^2$
  - $f(n) = O(g(n))$, i.e., $3n + 4 = O(n)$ because $3n + 4 \leq 7n\ \forall n \geq 1$ i. e., $|f(n)| \leq 7|g(n)| \forall n \geq 1$
  - $g(n) = O\big(h(n)\big), i.e., n = O(n^2)$ because $n \leq n^2\ \forall n \geq 1$
  - $f(n) = O\big(h(n)\big), i.e., 3n + 4 = O(n^2)$. (You can verify that $3n + 4 \leq 7n^2\ \forall n \geq 1$)
  - Question: which is preferable (more informative) to say:
    - $3n + 4 = O(n)$, or
    - $3n + 4 = O(n^2)$?

USE OF BIG-O IN COMPLEXITY ANALYSIS

1. Given an algorithm that you want to analyze
2. You derive an estimate of its time complexity, T(n), as a (possibly messy) expression of input size n
   - Example: $T(n) = 3n^{2.7} + n\sqrt{n} + 7n \log n$
3. View the $T(n)$ as your $f(n)$ and go find a much simpler function/expression $g(n)$ such that $f(n) = O(g(n))$

Choose the tightest and simplest $g(n)$ you can find

   - Example: for $T(n) = 3n^2 + n\sqrt{n} + 7n \log n$, take $g(n) = n^2$
   - Exercise: show that $3n^2 + n\sqrt{n} + 7n \log n \leq 11n^2$
   - As a result, you can conclude that $T(n) = O(n^2)$
   - Of course, one can also show that $T(n) = O(n^3)$

Which is better to say: $T(n) = O(n^2)$ or $T(n) = O(n^3)$? Why?

BIG-O
-- THEOREM TO HELP YOU FIND A GOOD $g(n)$ --

- **Theorem**: Let $f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_1 n^1 + a_0$ be a polynomial (in n) of degree m, where m is a positive constant integer, and $a_m, a_{m-1}, \ldots, a_0$ are constants. Then $f(n) = O(n^m)$.
- **Proof**:

$$
\begin{aligned}
|f(n)| &\le |a_m| n^m + |a_{m-1}| n^{m-1} + \cdots + |a_1| n^1 + |a_0| \\
&\le |a_m| n^m + |a_{m-1}| n^m + \cdots + |a_1| n^m + |a_0| n^m \\
&\le (|a_m| + |a_{m-1}| + \cdots + |a_1| + |a_0|) n^m \le k n^m
\end{aligned}
$$

where $k = |a_m| + |a_{m-1}| + \cdots + |a_1| + |a_0|$ and $n \ge 1$.

Therefore, by definition, $f(n) = O(n^m)$. Q.E.D.

BIG-O
-- A RULE OF THUMB TO FIND A GOOD $g(n)$ --

- General rule of thumb: If the time $T(n)$ is a sum of a **constant** number of terms, you can:
  - keep the largest-order term,
  - drop all the other terms,
  - drop the constant factor of the largest order term,
  - and thus get a simple Big-O form for $T(n)$.
- **Example**: If $T(n) = 3n^{2.7} + n\sqrt{n} + 7n \log n$, then $T(n) = O(n^{2.7})$. Why?
  - $n\sqrt{n} \leq n.n = n^2 \leq n^{2.7}$
  - $n \log n \leq n.n = n^2 \leq n^{2.7}$
  - Therefore: $3n^{2.7} + n\sqrt{n} + 7n \log n \leq 3n^{2.7} + n^{2.7} + 7n^{2.7} = 11n^{2.7}$, i.e., $|T(n)| \leq 11|n^{2.7}|\ \forall n \geq 1$
  - Therefore: $T(n) = O(n^{2.7})$
- **Question**: In the rule of thumb above, can you drop a **variable** number of terms?

BIG-O
-- TIME OF RECURSIVE ALGORITHMS --

- The time complexity of a recursive algorithm is often easier to calculate by:
  1. deriving a recurrence relation (i.e., express $T(n)$ in terms of $T(n-1)$ or $T(n/2)$ or *or* $T(m)$ for some $m < n$), and then
  2. solve the recurrence relation

- You will learn how to solve recurrence relations in this course

- Still, there is a theorem, the *Master Theorem*
  - helpful for solving recurrence relations that emerge in time complexity analysis of many recursive (e.g., divide and conquer) algorithms

BIG-O
-- THE MASTER THEOREM--

- **The Master theorem**: Let $a \geq 1$ and $b \geq 1$ be two constants, $f(n)$ a function, and $T(n)$ a function of non-negative n defined by the following recurrence relation:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ for } n > n_0$$

where n_0 is some constant, and the value of $T(n)$ for $n \leq n_0$ is $\leq$ some constant c. The precise values of those n_0 and c won't matter. Note that $\frac{n}{b}$ is taken to mean $\lfloor\frac{n}{b}\rfloor$ or $\lceil\frac{n}{b}\rceil$.

Then $T(n)$ has the following asymptotic bounds:

- If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
- If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant c<1 for all sufficiently large n, then $T(n) = \Theta\big(f(n)\big)$.

Please brush up on logarithms

By default, log is base 2: $\log n = \log_2 n$

HELPFUL FORMULAS FOR BIG-O

- **Stirling's Approximation**: $n! \cong \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, where e=2.718… is the base of natural logarithm
- Useful summation formulas:
  - $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$
  - $1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$
  - $1^3 + 2^3 + \cdots + n^3 = \left(\frac{n(n+1)}{2}\right)^2$
  - $1^k + 2^k + \cdots + n^k = O(n^{k+1})$, where k is a positive constant integer
  - $1 + x + x^2 + x^3 \ldots + x^n = \frac{x^{n+1}-1}{x-1}$, for all $x \neq 1$.
  - $1 + 2x + 3x^2 \ldots + nx^{n-1} = \frac{nx^{n+1}-(n+1)x^n+1}{(x-1)^2}$, for all $x \neq 1$.
  - $(a+b)^n = \binom{n}{n} a^n b^0 + \binom{n}{n-1} a^{n-1} b^1 + \binom{n}{n-2} a^{n-2} b^2 + \cdots + \binom{n}{k} a^{n-k} b^k + \cdots + \binom{n}{0} a^0 b^n$

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

PRACTICE EXERCISES

- **Exercise**: Prove the summation formulas (on the previous slide) by induction on n
- Note: don't turn in this exercise. Rather, it is just for your practice