INTRODUCTION

A. Preliminaries:

- Purpose: Learn the design and analysis of algorithms

- 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

- Functions and Procedures:
  - **Functions**: Algorithms that can be called by other algorithms and that return one output to the calling algorithm.
  - **Procedures**: Algorithms that can be called by other algorithms, and compute one or more outputs as side effect and/or as output parameters.

- Design of Algorithms:
  - Devising a method, using standard new techniques or standard existing techniques such as the ones covered in this course:
    - Divide and conquer
    - The greedy method
    - Dynamic programming
    - Graph traversal
    - Backtracking
    - Branch and bound
  - Expressing the algorithm (in a pseudo language, flowchart, high-level programming language, etc.)
  - Validating the algorithm (proof of correctness)

- Analysis: Determination of time and space requirements of the algorithm

- Implementation and Program Testing: outside the scope of this course.

B. Expression of Algorithms (Pseudo Language)

Notes: Words in **bold** are reserved words.

- 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;**
  Arrays: **int** A[1:n], B[4:10]; **char** C[1:n]; and the like.

- Assignments:

  X = Expression; or X := Expression; or X ← Expression;
  Examples: X = 1+3*4; Y=2*x-5; Z= Z+1;

- Control structures:

if *condition* **then**
a sequence of statements;
[else
a sequence of statements;]
endif
 	
Note: Things between brackets […] are optional
	if *condition1* **then**
	a sequence of statements;
	elseif *condition2* **then**
	a sequence of statements;
	…
	elseif *conditionk* **then**
	a sequence of statements;
	[else
	a sequence of statements;]
	endif

case x: *Value1***:** statements; [break;] *Value2***:** statements; [break;] **…** *Valuek***:** statements; [break;] **endcase**	**case:** *Cond1***:** statements; [break;] *Cond2***:** statements; [break;] **…** *Condk***:** statements; [break;] **endcase**

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

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

- Input-Output:

  read(X); // X is a variable or array or even an elaborate structure

  print(data); **write**(data, file); // data can numeric or strings

- Functions and Procedures:

function *name*(*parameters*)
begin
  variable declarations;
  sequence of statements;
  return (value);
end *name*
	procedure *name*(**input** *params;* **output** *params;*
	in-out *params*)
	begin
	  variable declarations;
	  sequence of statements;
	end *name*

- 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);
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
	end max

Procedure *swap*(**in-out x,y**)
Begin
  generic temp;
  temp=x;
  x=y;
  y=temp;
end swap

C. Recursion

- 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).
    name (smaller input); // this is a recursive call.
    // there can be more statements and more recursive calls here
    Combine subsolutions;
End
```

- 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 calling returning max of 1st half of the array
    y=max(A[m+1,j]); // recursive calling 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
```

D. Validation of Algorithms

- 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.

E. Analysis of Algorithms

- What it is: estimation of time and space (memory) requirements of the algorithm

- Why needed:
  - A priori estimation of performance to see if the conceived algorithm meets prior speed requirements (before any further investment of effort). If the algorithm is not fast enough, then the designer must come up with alternative (and faster) algorithms

  - A way for comparing algorithms. Sometimes one (or several competing designers) can design alternative algorithms for the same problem, and you need to determine which to choose. Typically the fastest algorithm (and/or least demanding in memory) is chosen.

- Machine Model:
  - Random access memory (RAM)
  - Arithmetic operations, Boolean operations, load/store read/write operations (of basic data types), and comparisons, take constant time each

- Time complexity $T(n)$: number of operations in the algorithm, as a function of the input size.

- Space complexity $S(n)$: number of memory words needed by the algorithm

- Example: the non-recursive max function takes time = n-1 comparisons, and space = 1.

- Since memory has become very cheap and abundant, we rarely care about space complexity. Time, however, is always a premium even if computers are always increasing in speed.

- For the purposes stated above, 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 slows down for very large input sizes, the time estimate can focus more on large input sizes n, and we thus should be more concerned about the "order of growth" of the time function $T(n)$, or as typically called, the asymptotic behavior of the $T(n)$.
    - Finally, 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 again 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 and will be provided next.

F. Asymptotics and Big-O Notation

- Big-O
    - Definition: let $f(n)$ and $g(n)$ be two functions of n (n is usually the input size in algorithm analysis). We say that
    $$f(n) = O(g(n))$$
    if $\exists$ an integer n_0 and a positive constant k such that
    $$|f(n)| \leq k|g(n)| \quad \forall n \geq n_0.$$
    - Example: $3n + 1 = O(n^2)$ since $3n + 1 \leq 3n^2 \quad \forall n \geq 2.\ n_0 = 2, k = 3.$
    - Example: $3n + 6 = O(n)$ because $3n + 6 \leq 4n \quad \forall n \geq 6.\ n_0 = 6, k = 4.$

- Big Omega (Ω)
  - 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
    - $|f(n)| \geq k|g(n)| \ \ \forall n \geq n_0$.
  - Example: $\frac{1}{3}n^2 = \Omega(n)$ because $\frac{1}{3}n^2 \geq n \ \ \forall n \geq 3$. $n_0 = 3, k = 1$.
  - Example: $3n + 6 = \Omega(n)$ because $3n + 6 \geq 3n \ \ \forall n \geq 1$. $n_0 = 1, k = 3$.

- Big Theta (Θ)
  - 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
    $$k_1|g(n)| \leq |f(n)| \leq k_2|g(n)| \ \ \forall n \geq n_0.$$
  - Example: $3n + 6 = \Theta(n)$ because $3n + 6 = O(n)$ and $3n + 6 = \Omega(n)$.

- **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: $|f(n)| \leq |a_m|n^m + |a_{m-1}|n^{m-1} + \cdots + |a_1|n^1 + |a_0|$
  $$\leq |a_m|n^m + |a_{m-1}|n^m + \cdots + |a_1|n^m + |a_0|n^m$$
  $$\leq (|a_m| + |a_{m-1}| + \cdots + |a_1| + |a_0|)n^m \leq kn^m,$$
  where $k = |a_m| + |a_{m-1}| + \cdots + |a_1| + |a_0|$ and $n \geq 1$. Therefore, by definition, $f(n) = O(n^m)$. Q.E.D.

- In general, if the time $T(n)$ is a sum of a constant number of terms, you can keep the largest-order term and drop all the other terms, and drop the constant factor of the largest order term, to get a simple Big-O form for $T(n)$.
  - Example: If $T(n) = 3n^{2.7} + n\sqrt{n} + 7n \log n$, then $T(n) = O(n^{2.7})$.
- The time complexity of a recursive algorithm is often easier to calculate by first deriving a recurrence relation (i.e., express $T(n)$ in terms of $T(n-1)$ or $T(n/2)$ *or* $T(m)$ for some $m < n$), and then solve the recurrence relation.

- You will learn how to solve recurrence relations in this course. Still, there is a theorem, the *Master Theorem*, which is very helpful for solving recurrence

relations that emerge in time complexity analysis of many recursive (e.g., divide and conquer) algorithms.

- **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: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ for $n > n_0$. (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 $T(n) = \Theta(f(n))$.

- Exercise: Apply the Master Theorem on the following problems to determine the order of $T(n)$:
  a. $T(n) = 6T\left(\frac{n}{3}\right) + n$
  b. $T(n) = 6T\left(\frac{n}{3}\right) + n^2$
  c. $T(n) = 6T\left(\frac{n}{3}\right) + n\sqrt{n}$
  d. $T(n) = 9T\left(\frac{n}{3}\right) + n^2$

- **Stirling's Approximation**: $n! \cong \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, where e=2.718…

- Some formulas useful in time complexity analyses (prove them by induction):
  - $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$
  - $1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$
  - $1^3 + 2^3 + \cdots + n^3 = \left(\frac{n(n+1)}{2}\right)^2$
  - $1^k + 2^k + \cdots + n^k = O(n^{k+1})$, 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 $x \neq 1$.
  - $(a+b)^n = \binom{n}{n}a^n b^0 + \binom{n}{n-1}a^{n-1}b^1 + \binom{n}{n-2}a^{n-2}b^2 + \cdots + \binom{n}{k}a^{n-k}b^k + \cdots + \binom{n}{0}a^0 b^n$