COSC160: Data Structures Review – Part 2

Jeremy Bolton, PhD

Assistant Teaching Professor

GEORGETOWN UNIVERSITY

Outline

I. Computational Complexity
   I. Time
      I. Upperbounding Notation
      II. Formalizing using Recurrences
   II. Space
      I. Recursion: Implications on Space Complexity

II. Implementation Details
   I. Pointers
   II. Memory Management

III. Good Programming Practices
   I. Design
   II. Debug

Review: Goals of Data Structures

- Computer science is the practice of problem solving.
  - Data structures are tools used to help solve a problem or accomplish a task.
  - Good Solutions (Algorithms):
    - Good solutions are correct.
    - Good solutions are practical.
    - Good solutions are efficient.
      - Efficient in time
      - Efficient in space

- Low level representations
  - Understanding lower level details related to memory management will make you a better programmer (of ***efficient*** structures and algorithms)

Computational Complexity Theory

- Algorithms on Data Structures are analyzed and assessed using two main factors
  - Time Complexity – number of “computational steps” required.
  - Space Complexity – number of memory “spaces” required.

- Algorithms may depend on size and value of input.
  - Value(s) of input
    - Cases: Worst Case, Best Case, Average Case.
  - Analysis in terms of size of input
    - Computational steps and memory requirements are generally a function of size of input.

- Analysis
  - “Absolute” count: Step count (time) or memory location (space) count
  - Count Upper-bound and lower-bound notation:
    - Big-O: upper bound
    - Big-Omega: lower bound
    - Big-Theta: both upper and lower bound

Best Case vs. Worst Case

- Cases are generally determined assuming the size of the input is fixed.
  - Best Case: Input (of size n) whose corresponding step count (or memory count) is minimum.
  - Worst Case: Input (of size n) whose corresponding step count (or memory count) is maximum.
  - Average Case: Average step count (or memory count) over all possible inputs (of size n).

- Therefore cases are often determined by the value of the input (not the size.)

- Example: Assume you have a list (of size n), and you are tasked with searching for an item in the list. What is the best case? Worst case?

Upper-bounding notation. (Formally)

- Big-O: f(x) is O(g(x)), when

$$\exists_{c\in\mathcal{R}^+,k\in\mathcal{R}^+}\forall_{x\geq k} f(x) \leq cg(x)$$

- Big-Omega: f(x) is Ω(g(x)), when

$$\exists_{c\in\mathcal{R}^+,k\in\mathcal{R}^+}\forall_{x\geq k} f(x) \geq cg(x)$$

- Big-Theta: f(x) is Θ(g(x)), when

$$\exists_{c_1\in\mathcal{R}^+,k_1\in\mathcal{R}^+}\forall_{x\geq k_1} f(x) \geq c_1 g(x)$$

and

$$\exists_{c_2\in\mathcal{R}^+,k_2\in\mathcal{R}^+}\forall_{x\geq k_2} f(x) \leq c_2 g(x)$$

Common Upper-bounding functions

g(x)= 1
g(x) = log(x)
g(x) = x log(x)
g(x) = x^2
g(x) = x^3
g(x) = e^x

Formalizing Time Complexity of An Algorithm

- Specify the step-count function by Explicit Counting.
  - If code is “simple”, count steps in the loop and determine how many times the loop will execute (depending on case.)
  - Use proof by induction (or direct arithmetic) to prove upperbound

- Recurrence.
  - Define step count function recursively.
  - Solve recurrence
    - By Iteration; or
    - By Substitution; or …
  - Use proof by induction (or direct arithmetic) to prove upperbound

Example: Computational Step Count

- Pseudocode for a sequential search algorithm below.
  - What is the number of computational steps in the worst case?
  - Often assumes all basic operations are 1 step.

```
// Searches sequentially for x in array. Returns index of first x, else -1     Step Count
Algorithm SequentialSearch (array, x){
        n := length(array);                                                    1
        index := -1;                                                           1
        for i := 1 to n do{                                                    3n + 2
                if x == array[i]{                                              n
                        index := i;                                            1?
                        break;                                                 1?
                }end if
        } end for
        return index;                                                          1
}end SequentialSearch
```

Loop overhead:	
i=1:	1 step (initial)
i <= n:	1 step (initial)
i++ (i=i+1):	2n steps (post-loop update)
i <= n:	n steps (loop re-entry)
Total:	3n+2

Total: $f(n) = 4n + (5 \text{ or so})$

$f(n)$ is $O(n)$

$f(n)$ is $\Omega(n)$

$f(n)$ is $\Theta(n)$

Bounding Proofs

- Explicit Counting Example: Prove f(n) = 4n+6 is O(n)
  - Proof Idea: We must find a c and a k such that the following inequality holds for all x greater than or equal to k
    - $\exists_{c\in\mathcal{R}^+,k\in\mathcal{R}^+}\forall_{n\geq k} f(n) \leq cg(n)$

- Once a reasonable c and k are chosen, this statement can be shown using a proof by induction.
  - Proof Idea: Choose c = 10, k = 1. We show $\exists_{c\in\mathcal{R}^+,k\in\mathcal{R}^+}\forall_{n\geq 1} 4n+6 \leq 10n, \text{by induction.}$
    - Base Case: Show for n = k. $4n+6 \leq 10n \equiv 10 \leq 10$ □
    - Induction: Assume $4n+6 \leq 10n$ and show $4(n+1)+6 \leq 10(n+1)$.

      $4n+6 \leq 10n$ (Fact: Assumption of inductive step)

      $4 \leq 10$ (Fact: This was chosen to help finish proof)

      $4n+4\ +6 \leq 10n+10$ (Addition of inequalities)

      $4(n+1)+6 \leq 10(n+1)$ □ (Factoring)

Example: Computational Step Count

- Pseudocode for a sequential search algorithm below.
  - What is the number of computational steps in the (non-pathological) best case?

```
// Searches sequentially for x in array. Returns index of first x, else -1     Step Count
Algorithm SequentialSearch (array, x){
        n := length(array);                                                    1
        index := -1;                                                           1
        for i := 1 to n do{                                                    2
                if x == array[i]{                                              1
                        index := i;                                            1
                        break;                                                 1
                }end if
        } end for
        return index;                                                          1
}end SequentialSearch
```

Total: $f(n) = 8$

$f(n)$ is $O(1)$

$f(n)$ is $\Omega(1)$

$f(n)$ is $\Theta(1)$

Example: Computational Step Count

- Pseudocode for a sequential search algorithm below.
  - What is the number of computational steps in the ***average case***?
    - Average case: take the average step count over all possible step count outcomes (cases)
    - $\frac{4n+5+\sum_{i=1}^{n} 4i+4}{n} = \frac{4n+5+\sum_{i=1}^{n} 4i+\sum_{i=1}^{n} 4}{n} = \frac{4n+5+4\sum_{i=1}^{n} i+4n}{n} = \frac{8n+5+4\frac{n(n+1)}{2}}{n} = 8 + \frac{5}{n} + 2(n+1)$, which is Θ(n)

```
// Searches sequentially for x in array. Returns index of first x, else -1
Algorithm SequentialSearch (array, x){
        n := length(array);
        index := -1;
        for i := 1 to n do{
                if x == array[i]{
                        index := i;
                        break;
                }end if
        } end for
        return index;
}end SequentialSearch
```

Tip: organize different step counts in table, then take average.

Total: f(n) = $\frac{3}{n} + 4 + 4\,[(n+1)/2)]$

f(n) is O(n)

f(n) is Ω(n)

f(n) is Θ(n)

Case	Count
1	8
2	12
3	16
…	… note: arithmetic sequence α_n
n	$\alpha_n = 4n + 4$
n+1 (not in list)	4n+5

More Examples: Bounding Notation

- Assume you have analyzed an algorithm and determined that the step count (in the worst case) is f(x) = 10x +7, for inputs of size x.
  - Find Big Theta Notation (that is find g(x) such that f(x) is Θ(g(x)))
  - Big-Theta: f(x) is Θ(g(x)), when

$$\exists_{c_1 \in \mathcal{R}^+, k_1 \in \mathcal{R}} \forall_{x \geq k_1} f(x) \geq c_1 g(x)$$

and

$$\exists_{c_2 \in \mathcal{R}^+, k_2 \in \mathcal{R}} \forall_{x \geq k_2} f(x) \leq c_2 g(x)$$

- Tip: The largest term in this function is linear, therefore we can bound f(x) proportionally with a linear function: g(x) = x.

Notes about bounding notation

- Big-O notation: gives us the upper bound.
  - Somewhat informative*

- Big-Omega: lower bound
  - Somewhat informative*

- Big-Theta: identifies the order by which the algorithm will scale (as n become large.)
  - Most informative*

Quickly assess the time complexity

– **<u>Worst Case</u>**: $cn + k + 3$, is $\Theta(n)$

```
// Searches sequentially for x in array. Returns index of first x, else -1     Step Count
Algorithm SequentialSearch (array, x){
        n := length(array);                                                    1
        index := -1;                                                           1
        for i := 1 to n do{                                                    cn+k
                if x == array[i]{
                        index := i;
                        break;
                }end if
        } end for
        return index;                                                          1
}end SequentialSearch
```

Assuming a constant number of operations per iteration, then the total number of steps is of the form *cn+k*, where c and k are constants and n is the number of iterations

Time Complexity using Recurrences

- Recursion
  - Defining an entity in terms of itself

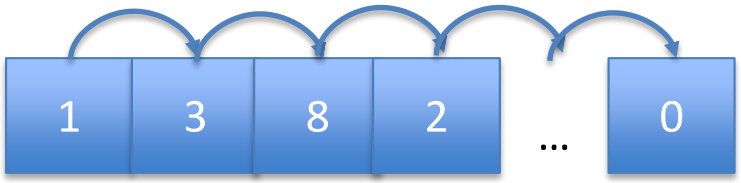
- Recurrence
  - A relation that characterizes a function in terms of its value on a smaller input

- General Idea:
  - Many computer science solutions can be defined recursively
  - Similarly, many step count functions can be defined using a recurrence

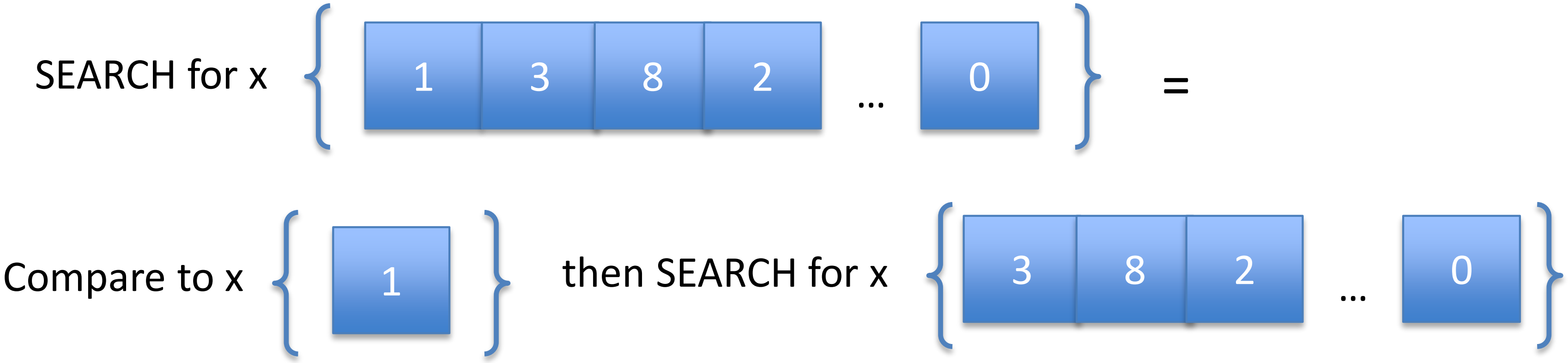
Defining Sequential Search Recursively

- Problem: search a list X of length n for some value x

- Iterative Perspective

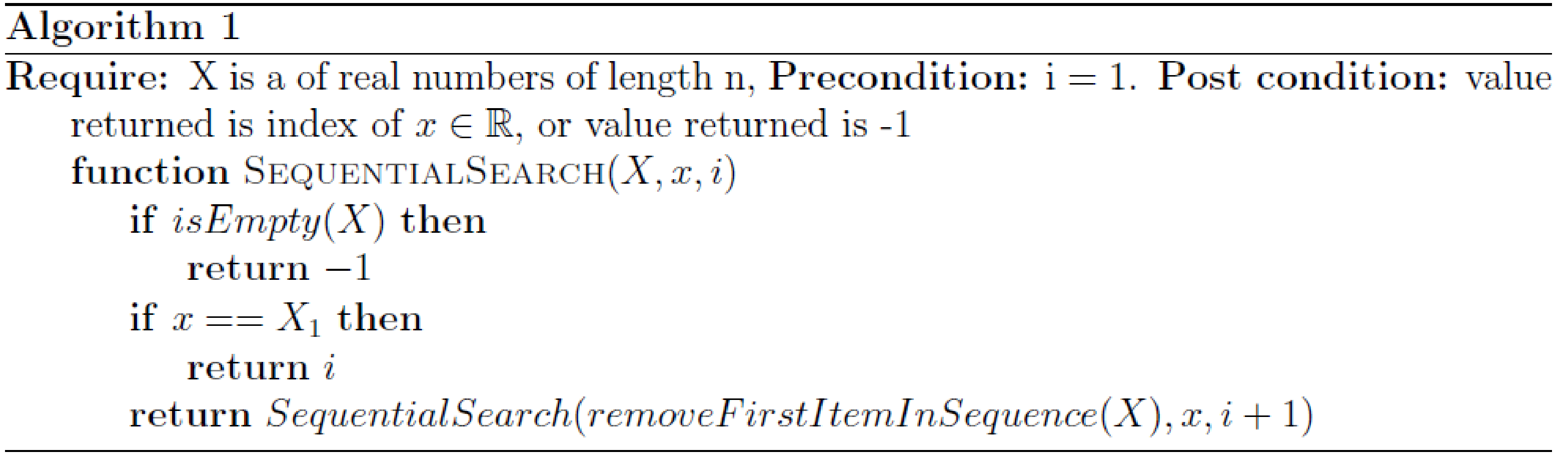


- Recursive Perspective


SEARCH for x { 1 3 8 2 … 0 } =

Compare to x { 1 } then SEARCH for x { 3 8 2 … 0 }

Time Complexity using Recurrence Example: Sequential Search



```
Algorithm 1
Require: X is a of real numbers of length n, Precondition: i = 1. Post condition: value
    returned is index of $x \in \mathbb{R}$, or value returned is -1
    function SequentialSearch($X, x, i$)
        if $isEmpty(X)$ then
            return $-1$
        if $x == X_1$ then
            return $i$
        return $SequentialSearch(removeFirstItemInSequence(X), x, i + 1)$
```

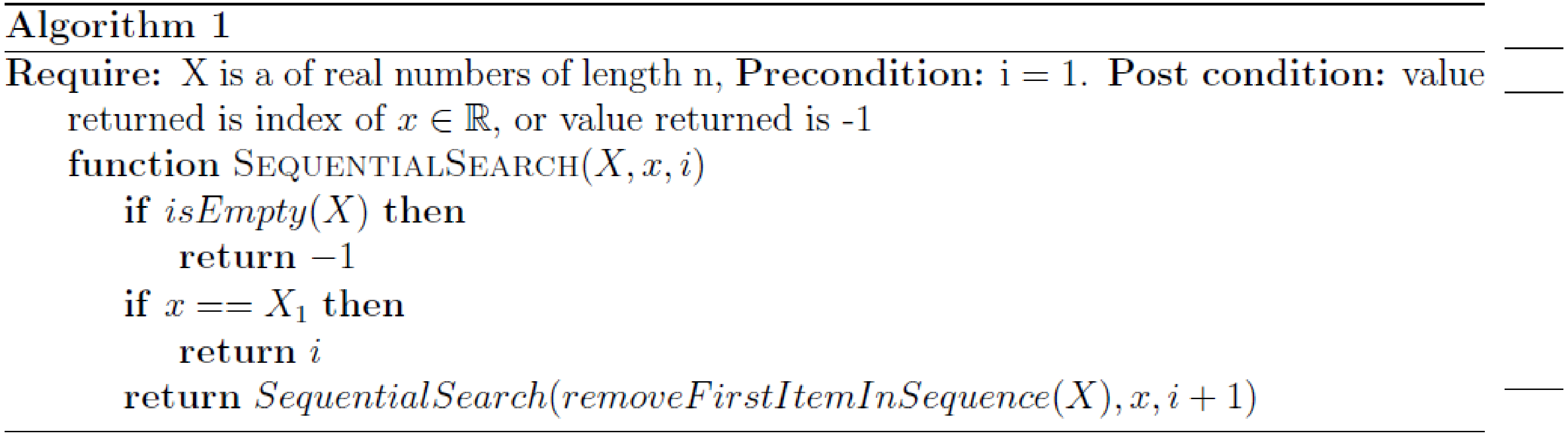
- Assess Complexity by Using Recurrence and Induction (using substitution)
  1. Define algorithm and then step count recursively
     - Search(list,x,n) := compare(list[1],x) + Search(remainingList,x,n-1)
       – Where remainingList is list less the first item
     - T(n) := 1 + T(n-1) , where initial condition T(1) = 1
  2. “Guess” an upperbound
     - E.G. I guess T(n) is O(n)
  3. Prove it using induction
     - Base Case
     - Inductive Case

T: recursively defined step count function. T(n) is the total number of comparisons count for searching a list of length n

Time Complexity using Recurrence (substitution) Example: Sequential Search

- Show $T(n) \leq cn$, for some c.
- Proof:
  - Inductive Case. Assume conjecture is true for n-1, show holds for n
    - $T(n-1) \leq c_1(n-1)$ — Assumption 1
    - $T(n) = T(n-1) + 1$ — Assumed Recurrence
    - $T(n) = T(n-1) + 1 \leq c_1(n-1) + 1$ — Using Assumption 1
    - $T(n) \leq c_1(n-1) + 1 \leq c_1 n$ — for some $c_1 > 1$, when n > 0
    - $T(n) \leq c_1 n - c_1 + 1 \leq c_1 n$
    - $T(n) \leq c_1 n$ $\square$
  - Base Case (n=1)
    - $T(1) = 1 \leq c_1 1$ — for some c_1 >1

Time Complexity using Recurrence Example: Sequential Search



```
Algorithm 1
Require: X is a of real numbers of length n, Precondition: i = 1. Post condition: value
    returned is index of $x \in \mathbb{R}$, or value returned is -1
    function SEQUENTIALSEARCH($X, x, i$)
        if $isEmpty(X)$ then
            return $-1$
        if $x == X_1$ then
            return $i$
        return $SequentialSearch(removeFirstItemInSequence(X), x, i + 1)$
```

- Assess Complexity by *Solving Recurrence* (using iteration)
  1. Define algorithm and then step count recursively
     - T(n) := 1 + T(n-1) , where initial condition T(1) = 1
  2. Enumerate terms in recurrence (forward or backward) until the boundary condition
  3. Identify sequence or series and use knowledge of series to compute closed form of the total step count. (Solve the recurrence)
  4. Upperbound if desired.

GEORGETOWN UNIVERSITY

Time Complexity using Recurrence Example: Sequential Search

- Enumerate to boundary (here we use backward substitution)

$$\begin{aligned} T(n) &= T(n-1)+1 \\ &= T(n-2)+1+1 \\ &= T(n-3)+1+1+1 \\ &= \cdots \\ &= T(1)+1+1+\cdots+1 \\ &= 1+1+1+\cdots+1 \end{aligned}$$

T(n-1) = T(n-2) + 1

…

…

Sum of n terms

Searching 1 item is 1 step

- Identify sequence.
  - This is an arithmetic sequence (series)
    - $T_{\mathrm{n}} = \mathrm{T}_{\mathrm{n}-1} + 1$
    - $T_n = \sum_{i=1}^{n} 1 = \mathrm{n} \leq cn$

For some c

GEORGETOWN UNIVERSITY

Example: Memory Requirements

- Space Complexity

```
// Searches sequentially for x in array. Returns index of first x, else -1          Memory Count
Algorithm SequentialSearch (array, x){                                              n+1
        n := length(array);                                                         1
        index := -1;                                                                1
        for i := 1 to n do{                                                         3
                if x == array[i]{                                                   1
                        index := i;
                        break;
                }end if
        } end for
        return index;
}end SequentialSearch
```

Example: Selection Sort

- Selection Sort (conceptually)
  1. Scan through oldList and find min item
  2. Remove minItem from oldList and place it next spot at the next index in orderedList
  3. Repeat

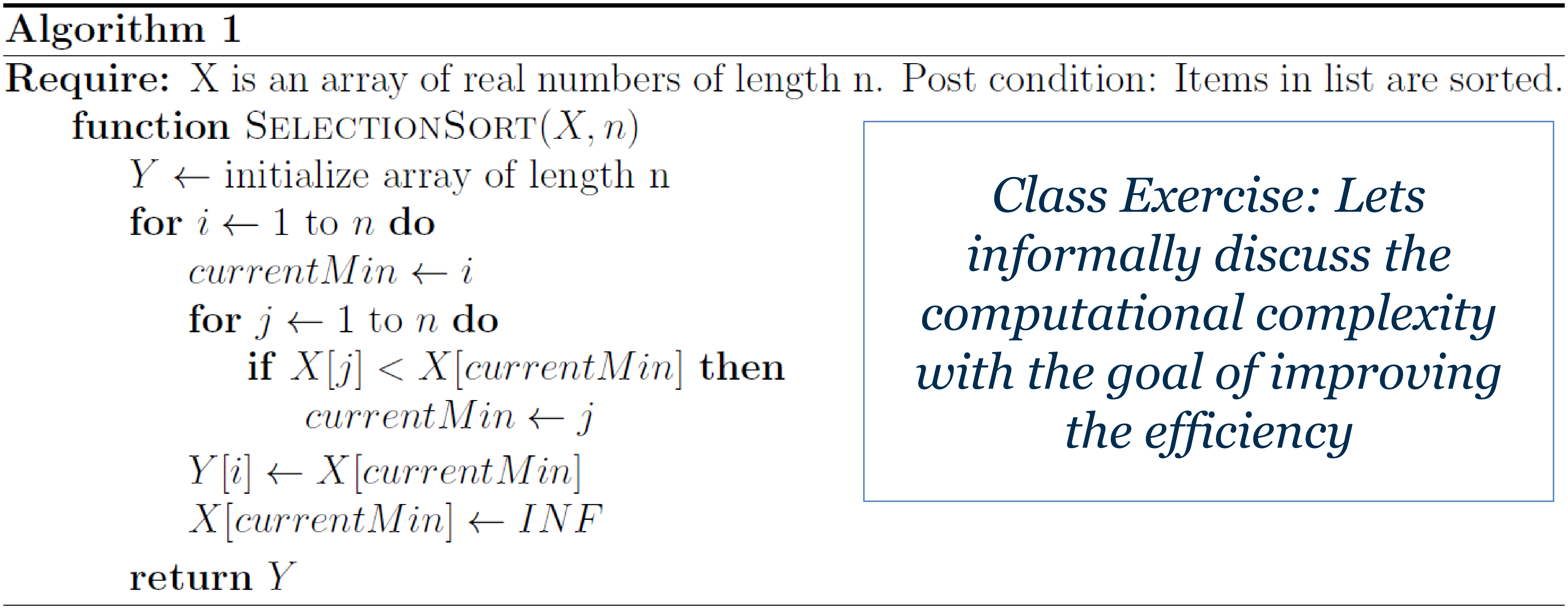
- Example
  1. Lets formalize this algorithm (using pseudocode)
  2. Lets assess the computational complexity

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Recall: Definition of an Algorithm

- PROPERTIES OF ALGORITHMS
  - **Input domain** is defined
  - **Output domain** is specified
  - The output or result is **correct**
    - Preconditions: logical statements assumed to be true prior to algorithm execution
    - Postconditions: logical statements assumed to be true after algorithm execution
  - The constituent steps are **well-defined**
  - Each step is **performable** in **finite time**

Sorting Example: Selection Sort
Assess Time and Space Complexities



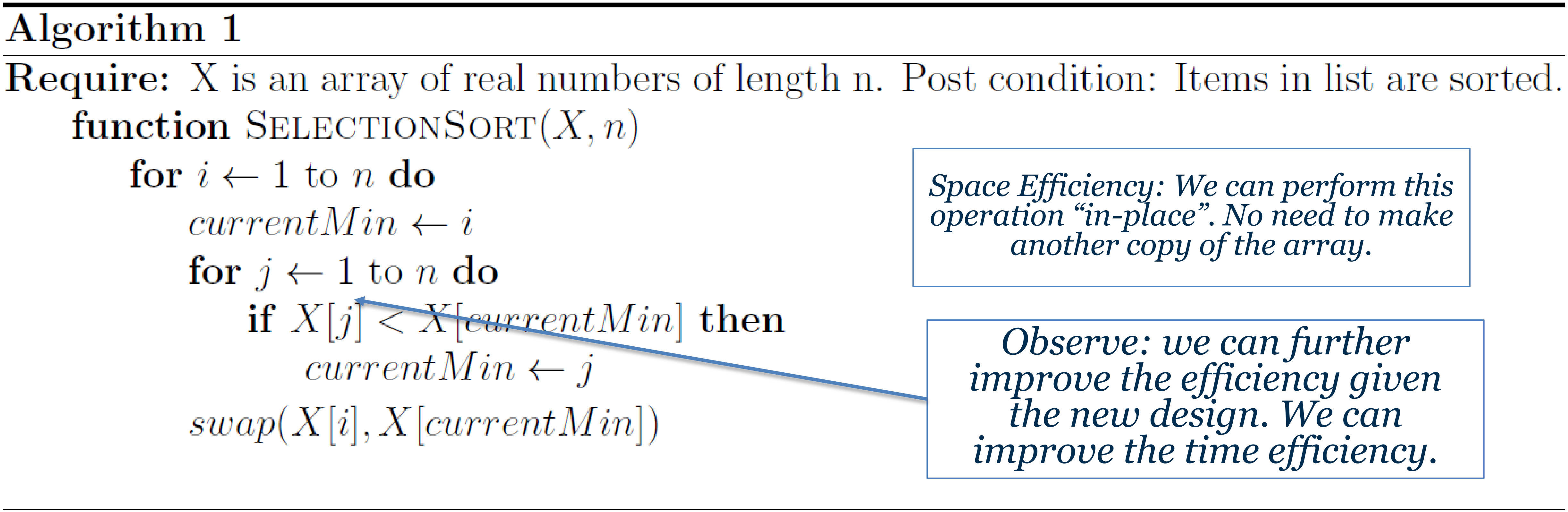
Algorithm 1

Require: X is an array of real numbers of length n. Post condition: Items in list are sorted.

```
function SelectionSort($(X, n)$)
    $Y \leftarrow$ initialize array of length n
    for $i \leftarrow 1$ to $n$ do
        $currentMin \leftarrow i$
        for $j \leftarrow 1$ to $n$ do
            if $X[j] < X[currentMin]$ then
                $currentMin \leftarrow j$
        $Y[i] \leftarrow X[currentMin]$
        $X[currentMin] \leftarrow INF$
    return $Y$
```

Class Exercise: Lets informally discuss the computational complexity with the goal of improving the efficiency

Sorting Example: Selection Sort
Assess Time and Space Complexities



Algorithm 1

Require: X is an array of real numbers of length n. Post condition: Items in list are sorted.

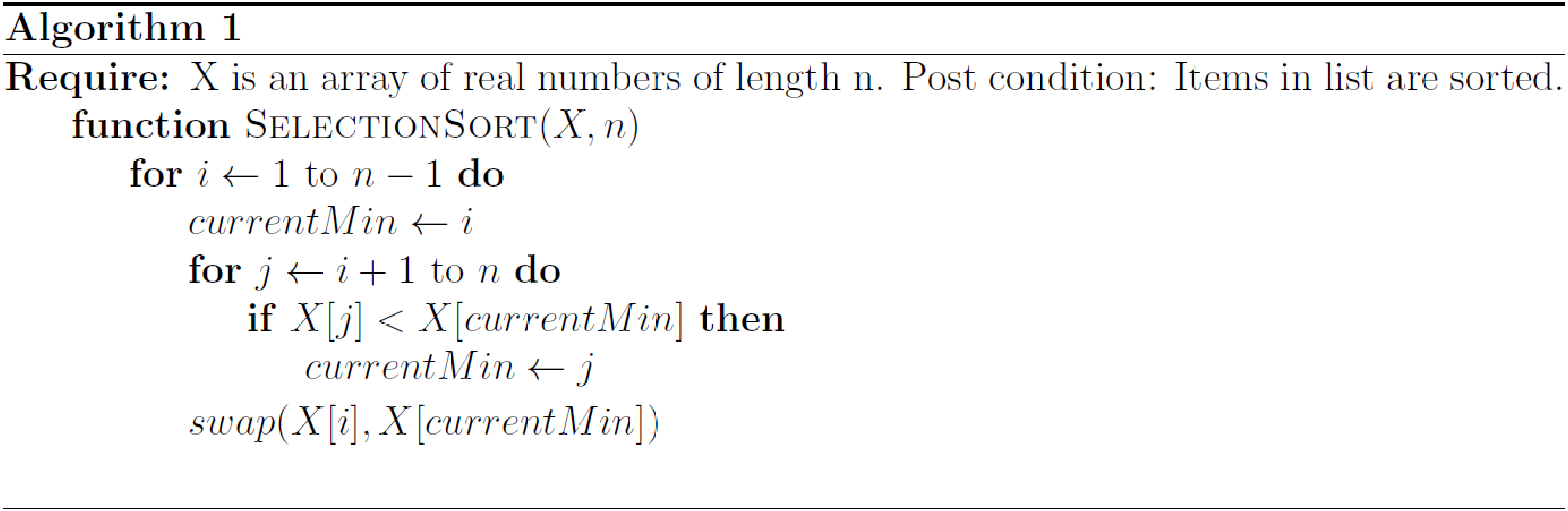
```
function SelectionSort($(X, n)$)
    for $i \leftarrow 1$ to $n$ do
        $currentMin \leftarrow i$
        for $j \leftarrow 1$ to $n$ do
            if $X[j] < X[currentMin]$ then
                $currentMin \leftarrow j$
        $swap(X[i], X[currentMin])$
```

Space Efficiency: We can perform this operation "in-place". No need to make another copy of the array.

Observe: we can further improve the efficiency given the new design. We can improve the time efficiency.

GEORGETOWN UNIVERSITY

Sorting Example: Selection Sort
Assess Time and Space Complexities



Algorithm 1

Require: X is an array of real numbers of length n. Post condition: Items in list are sorted.

```
function SelectionSort($X, n$)
    for $i \leftarrow 1$ to $n - 1$ do
        $currentMin \leftarrow i$
        for $j \leftarrow i + 1$ to $n$ do
            if $X[j] < X[currentMin]$ then
                $currentMin \leftarrow j$
        $swap(X[i], X[currentMin])$
```

Practical vs Theoretical Complexity Analysis

- Scenario: Assume two algorithms with 2 different step count functions
  - f_1(n) = 4n+10
  - f_2(n) = 400n + 10
- The theoretical time complexity is similar $\Theta(n)$
  - But for practical purposes, it is quite notable that algorithm 1 may be ~100 times faster!
- Keeping track of the lower level details is important in practice.

Other ***Practical Notes*** *about Computational Complexity*

- The process of theoretically analyzing algorithms is (over) simplified.
  - Differences in data types may be ignored
    - BUT some different data types have significantly different memory requirements and computational requirements
  - Differences in operations may be ignored
    - BUT not all operations require the same amount of runtime
      - Bit-wise operator vs. addition (of ints) vs multiplication (of floats)
  - Details in implementation / application may be significant in practice!

The Devil is in the Details

- Time
  - Not all basic operations have the same computation time!

- Memory
  - Not all data types take up the same amount of memory!
  - Increase in memory used will generally increase computation time!
  - Details of implicit memory management details can have drastic effects on space (and time) complexity
    - Pass by value vs pass-by-reference
    - Deep vs shallow copy
  - Memory Constraints
    - Virtual memory – hard disk retrieval
    - Memory capacity and paging

- These details will affect complexity counts (and possibly the order of complexity)
  - Using operators and memory management schemes smartly can drastically improve efficiency

Outline

- I. Computational Complexity
  - I. Time
    - I. Upperbounding Notation
    - II. Formalizing using Recurrences
  - II. Space
    - I. Recursion: Implications on Space Complexity

- II. Memory Management and Implementation Details
  - I. Pointers
  - II. Allocation on Stack vs Heap

- III. Good Programming Practices
  - I. Design
  - II. Debug

Memory Requirements

- Counting memory requirements of an algorithm
  - Where is the data allocated … and who cares!
    - Stack vs heap
      - A peak into the stack
      - Recursion and memory requirements

- Memory requirements of data structures
  - Indigenous data vs exogenous data
  - Deep copy vs shallow copy

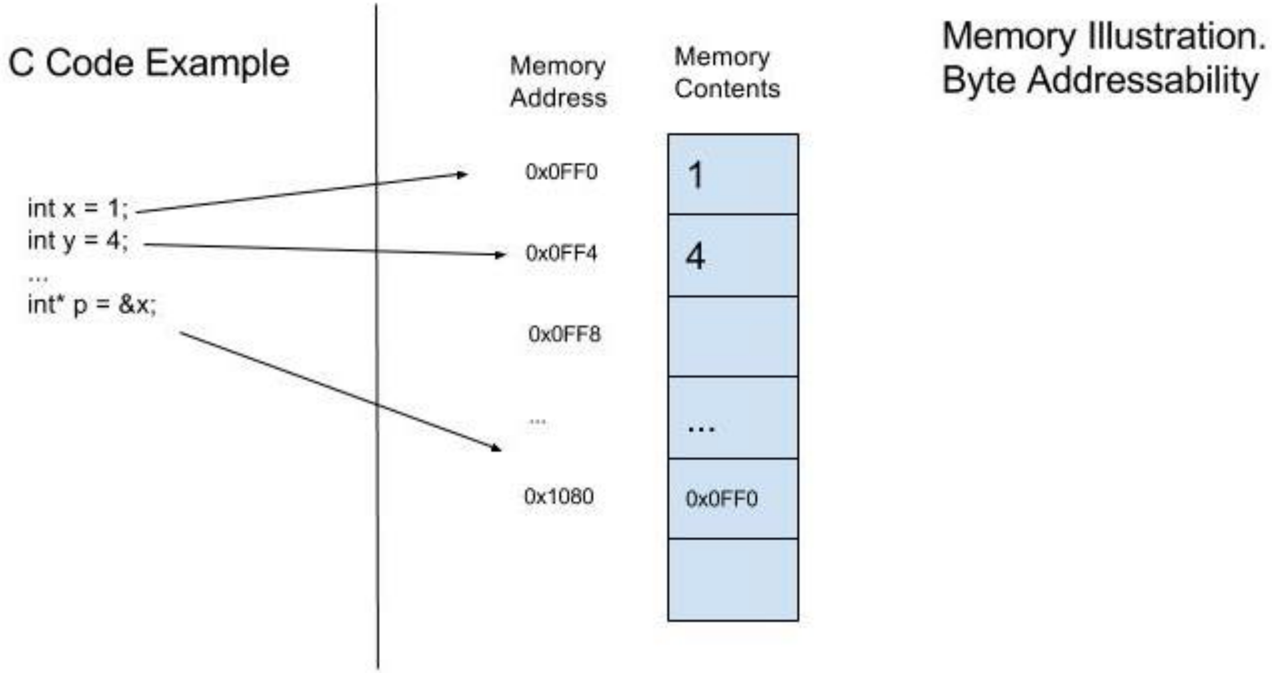
- Memory Notes:
  - Theoretical analysis: some include size of input, some do not
  - Practical analysis: are parameters pass by value vs. pass by reference
    - The parameters are stored somewhere in memory, but are they going to be copied onto the stack. If so, then you should certainly include the space in your memory requirements
    - When should you pass-by-copy?

Basic Designs for Structures

- Contiguous vs non-contiguous memory allocation
  - Arrays
    - Size does not change.
    - Data stored **contiguously** (in memory) in computer.
    - Data can be *directly* accessed.
  - Chaining (eg linked lists)
    - Size can change dynamically
    - Data may **not be stored contiguously** in memory.
    - Data is accessed only by traversing the list.

Review: Pointers

- Pointers are variables that contain the address of another variable
  - Allows for efficient processing of large structures
  - Example:
    - Assumes 4 byte ints



```
int x = 1;
int y = 4;
...
int* p = &x;
```

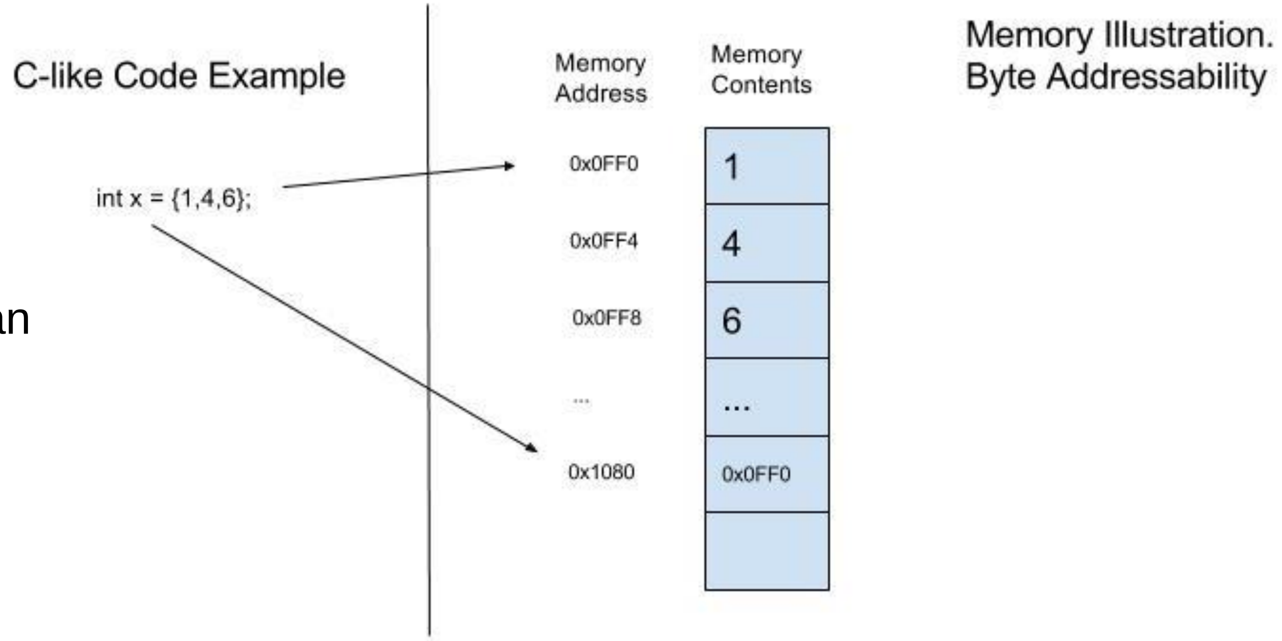
Memory Address	Memory Contents
0x0FF0	1
0x0FF4	4
0x0FF8	
...	...
0x1080	0x0FF0

GEORGETOWN UNIVERSITY

Arrays in Memory

- An array is a list of data stored contiguously in memory.
  - Its contiguous nature is both an advantage and a disadvantage.
    - Pro: items in the list can be quickly accessed given the base address and an offset
    - Con: memory must allocated (reserved) when the array is created. As a result, the size cannot change.

- In C++, array names are pointers to the base address of the array
  - Bracket Notation
    - x[1] evaluates to 2
  - Pointer Notation
    - *(x+1) evaluates to 2



C-like Code Example

```
int x = {1,4,6};
```

Memory Illustration.
Byte Addressability

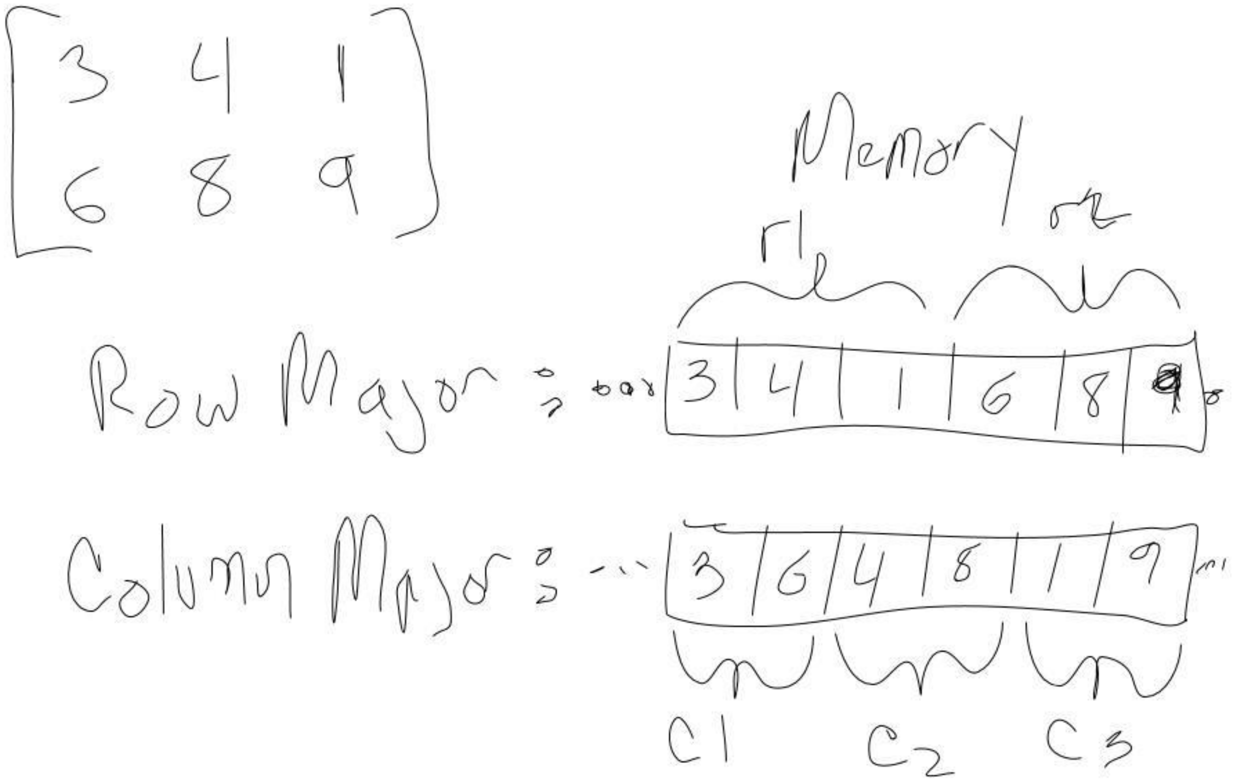
Memory Address	Memory Contents
0x0FF0	1
0x0FF4	4
0x0FF8	6
...	...
0x1080	0x0FF0

Multi-dimensional Arrays in Memory

- Multi-dimensional arrays are stored in memory similarly as one-dimensional arrays
  - Memory can be seen as a linear structure.

- Row-Major: rows are stored in sequence

- Column-Major: columns are stored in sequence

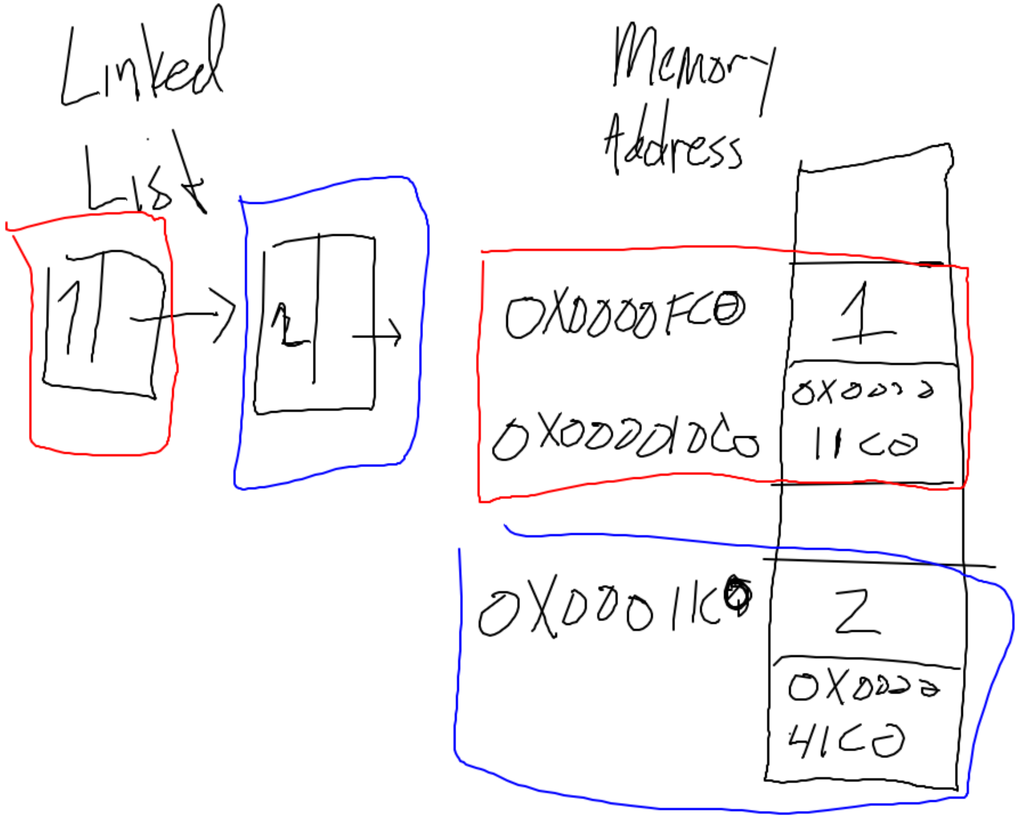


Lists using “Chaining”

- Some lists can change size once created. This is generally accomplished by chaining.
  - The data cannot necessarily be stored contiguously in memory in this instance (as the memory is not allocated at one time.) Thus the data may not be sequentially contiguous in memory.
    - But then how can we can we access items in the list?
    - This extra flexibility comes at the cost of storing 1 pointer per data item stored. This pointer will point to the next item in the list, thus facilitating a “linked” list.
- A linked (or chained) structure in memory

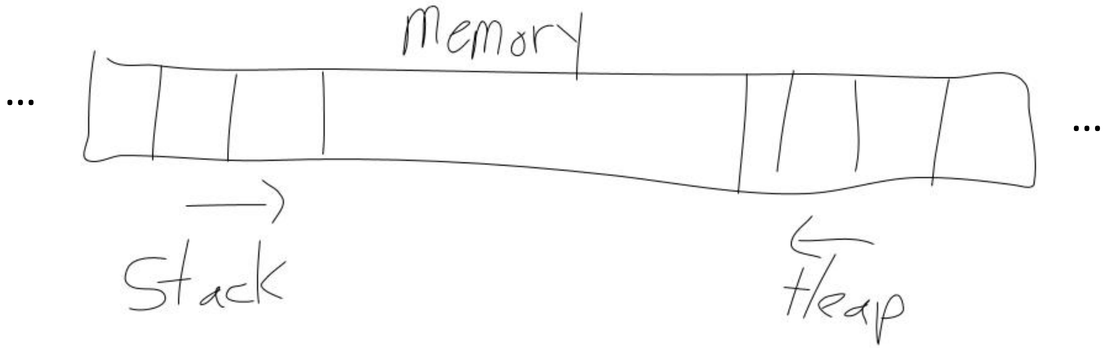
Linked Lists in Memory

- A linked list example
  - Each node is stored in memory, but subsequent nodes are not necessary contiguous in memory. The “next” field is used to point to the next item in the list. The next field is simply a pointer stored in memory.



Memory Allocation (in practice)

- Memory: Stack vs. Heap
  - Most local variables are allocated on the runtime stack.
    - The stack is largely “self-managing”.
  - If the size of a data structure is **not** known at compile time, it can be allocated on the heap.
    - Often the onus of memory management falls on the programmer
  - Memory Management: Managing your data (and structure)
    - The run-time stack largely manages itself
    - The heap may require explicit deallocation (Garbage Collection).
      - Memory Leaks.
      - Dangling Pointers

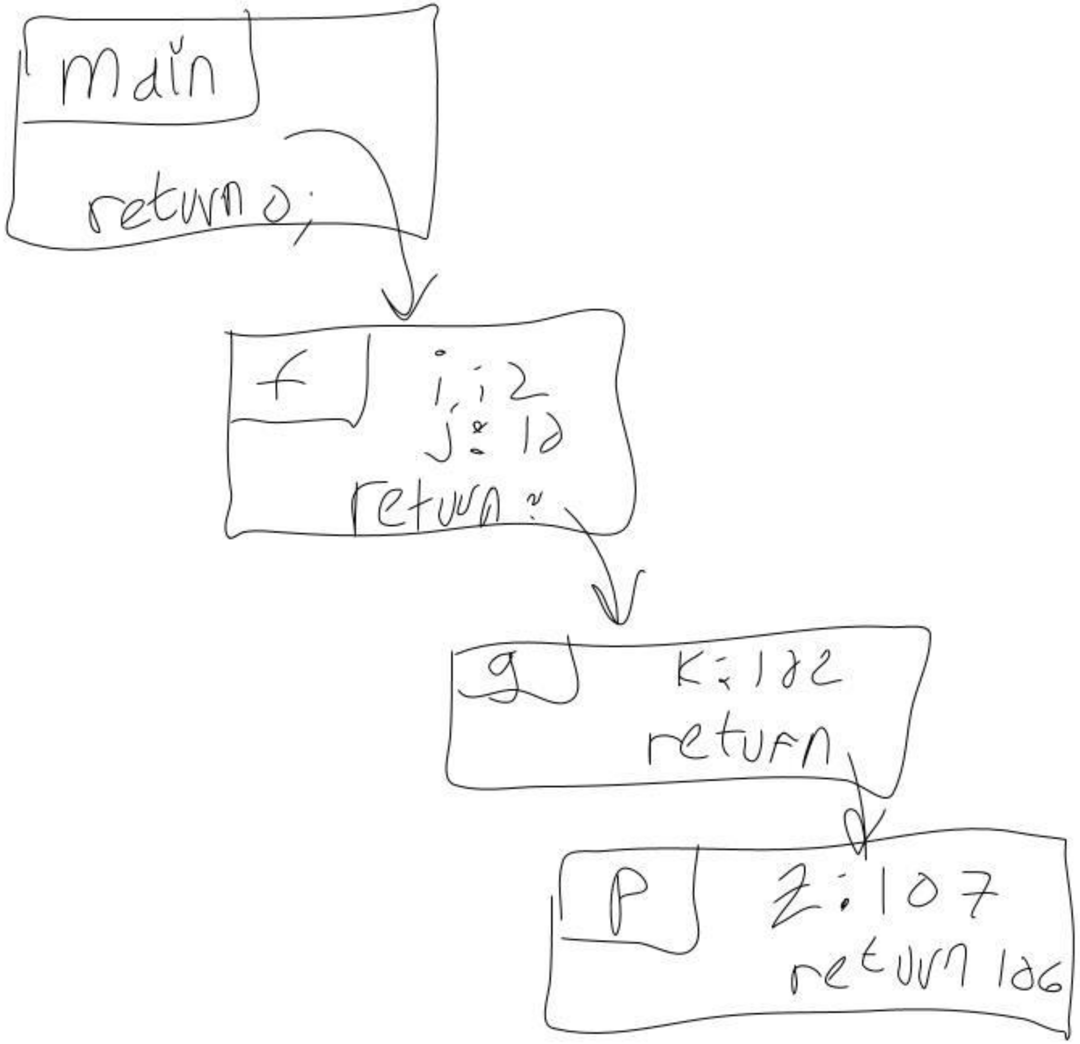


Memory and the Runtime Stack (in practice)

- In most programming languages, when a new scope is opened, (e.g. a function is called), a new frame of memory is reserved for all variables local to that scope (as well as some memory other data / variables such as input parameters, return values, intermediate calculations, …).
- Each scope or frame of memory is reserved on a dynamically-allocated, *contiguous* portion of memory referred to as the runtime stack.
  - Function call chains
  - Recursion example

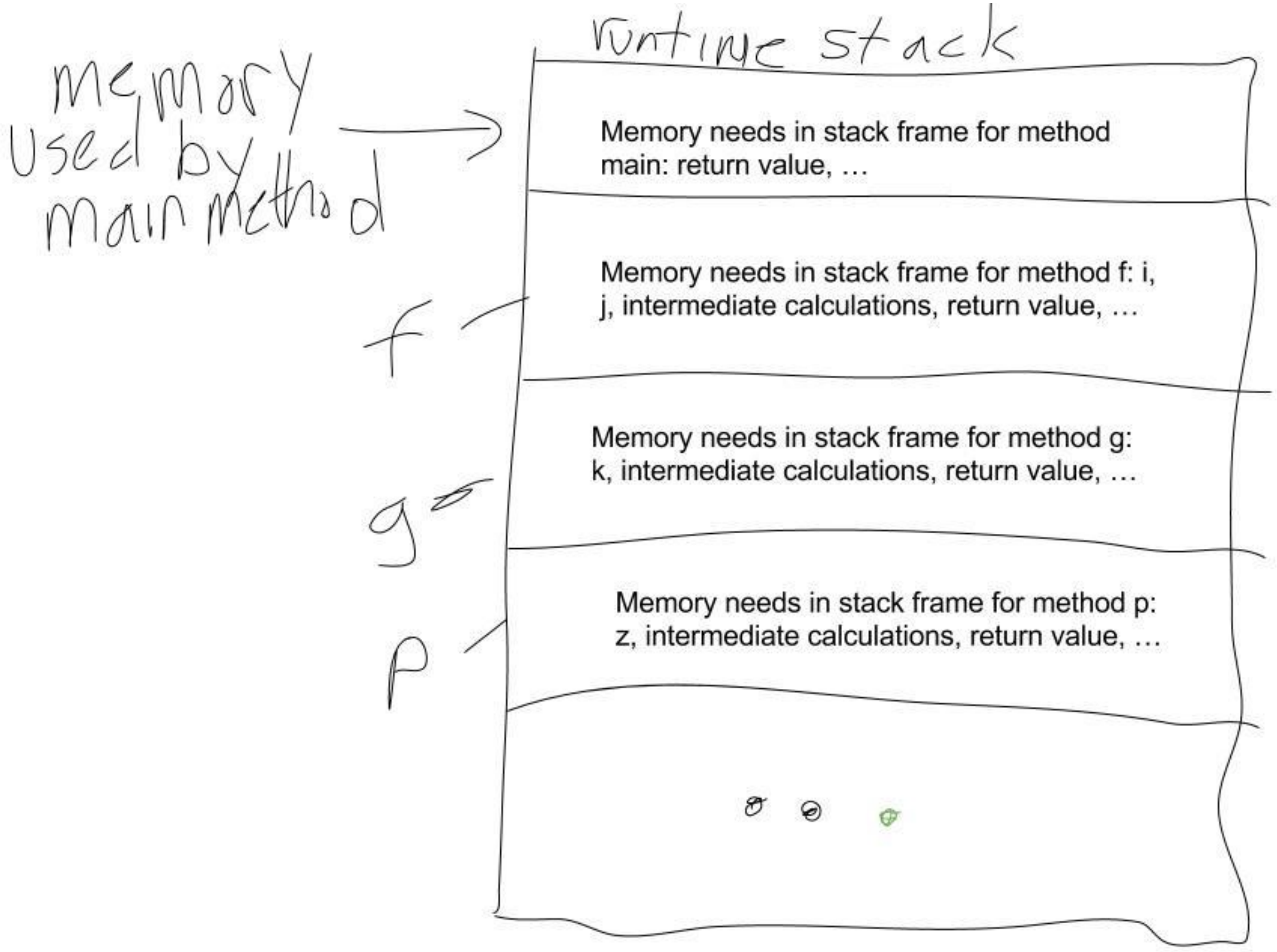
Scope Diagram: A common way to illustrate function call chains (flow of execution and variable values)

```
int f(int i)
    {
    int j = 10;
    return g(i + 10 * j);
    }

int g(int k)
    {return p(k + 5);}

int p(int z)
    {cout << z;
    return z - 1;
    }

int main()
{ f(2); return 0; }
```



Function call chain and the runtime Stack

```
int f(int i)
    {
    int j = 10;
    return g(i + 10 * j);
    }

int g(int k)
    {return p(k + 5);}

int p(int z)
    {cout << z;
    return z - 1;
    }

int main()
{ f(2); return 0; }
```



Memory needs in stack frame for method main: return value, …

Memory needs in stack frame for method f: i, j, intermediate calculations, return value, …

Memory needs in stack frame for method g: k, intermediate calculations, return value, …

Memory needs in stack frame for method p: z, intermediate calculations, return value, …

Recursion

- Recursive methods can greatly impact memory allocation

- Recursive methods are methods that are defined in terms of themselves.

- Many processes and calculations that involve repetition can be implemented recursively.

Recursion Example: Factorial

- Write a function to calculate n! where n is the input argument

  n! = n*(n-1)*(n-2)*… *2*1

  n! = n*(n-1)!

  factorial(n) =n*factorial(n-1)

- Note that this calculation is inherently repetitive.
  - There is repeated multiplication to be performed
  - Further note that the number of multiplications is dependent on “n” that is the input parameter will control the number of repetitive computations (multiplications in this case)

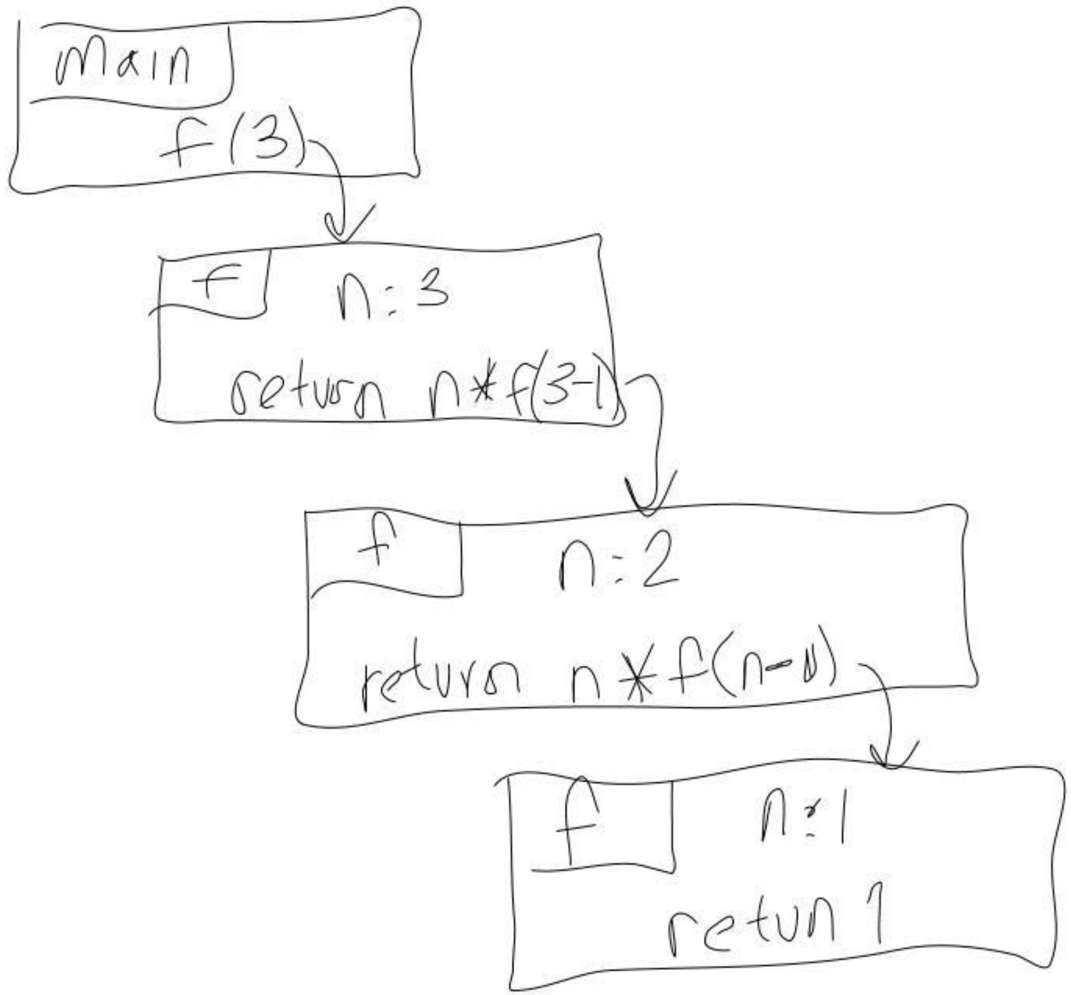
Factorial Examples

```
int factorial(int n)
    {
    //Assumes non-negative n
    int val = 1;
    for (int i = n; i > 1; i--;) //
repeatedly take product of values
between 1 and n
        val = val * i;

    return val;
    }
```

```
int factorial(int n)
    {
    // Assumes n is non-negative
    int val = 1;
    if (n == 0 || n == 1) // Base case --
stop repetition
        return 1;
    else  // recursive case -- continue
recursive call
    return n * factorial(n - 1);
    }
```

Draw Scope (function chain) diagrams to trace the execution of a recursive function.

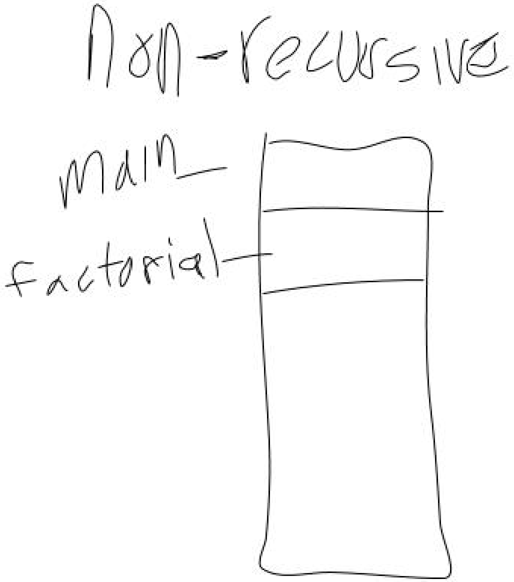
```
int f(int n)
       {
       // Assumes n is non-negative
       int val = 1;
       if (n == 0 || n == 1) // Base case -- stop
repetition
              return 1;
       else  // recursive case -- continue
recursive call
       return n * f(n - 1);
       }

void main()
{f(3);}
```

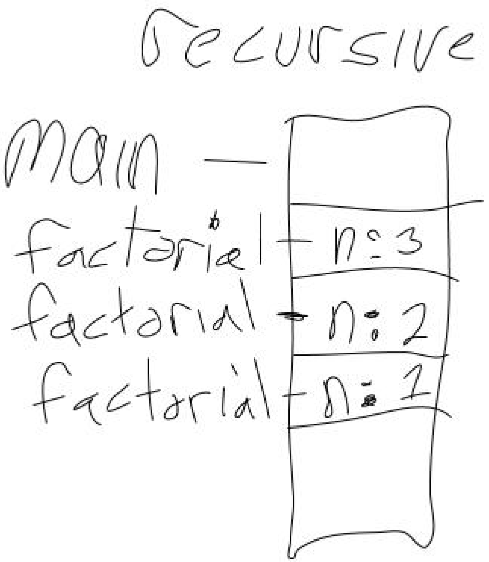


Recursion (and non-recursion) and the Stack

```
int factorial(int n)
     {
     //Assumes non-negative n
     int val = 1;
     for (int i = n; i > 1; i--;) //
repeatedly take product of values between 1
and n
          val = val * i;

     return val;
     }
```

```
int factorial(int n)
     {
     // Assumes n is non-negative
     int val = 1;
     if (n == 0 || n == 1) // Base case --
stop repetition
          return 1;
     else  // recursive case -- continue
recursive call
     return n * factorial(n - 1);
     }
```





In-Class (or at-home) Example

- Recursion
  - int sum(i , n) : computes sum of ints from i to n
  - int exp(x , n) : computes x^n

- Define the algorithm recursively

- Provide c-like code

- Analysis: Define a step (or space) count function as a recurrence. Then determine the time (or space) complexity.

Heap Allocation, basic memory management, garbage collection

- If the size (memory needed) of a variable (data) is known at compile time, it can be allocated on the stack; otherwise, if the size of the variable is not known at compile time, it can be allocated on the heap.

- Variables allocated on the heap are accessed using pointers. Allocation is on the heap is sometimes explicitly declared syntactically. Example: C++ and the keyword “new”.

- Data allocated on the heap *may be* explicitly deallocated (collect the garbage). Example: C++ and the keyword “delete”.

Common Memory Management Issues

- Dangling Pointer
  - A structure in memory has been deallocated, but the pointer still points to this place in memory.
    - Problem: pointer points to some place in memory, but the data there may no longer be valid.
    - Corrective Measure: Set pointer to null to clearly indicate pointer points nowhere.
- Memory Leak
  - When a pointer no longer points to some structure in memory, and the structure has not been deallocated.
    - Problem: Data is stored in memory, however it can no longer be accessed (or explicitly deleted!!!)
    - Corrective Measure: Pray to the garbage-collection gods.

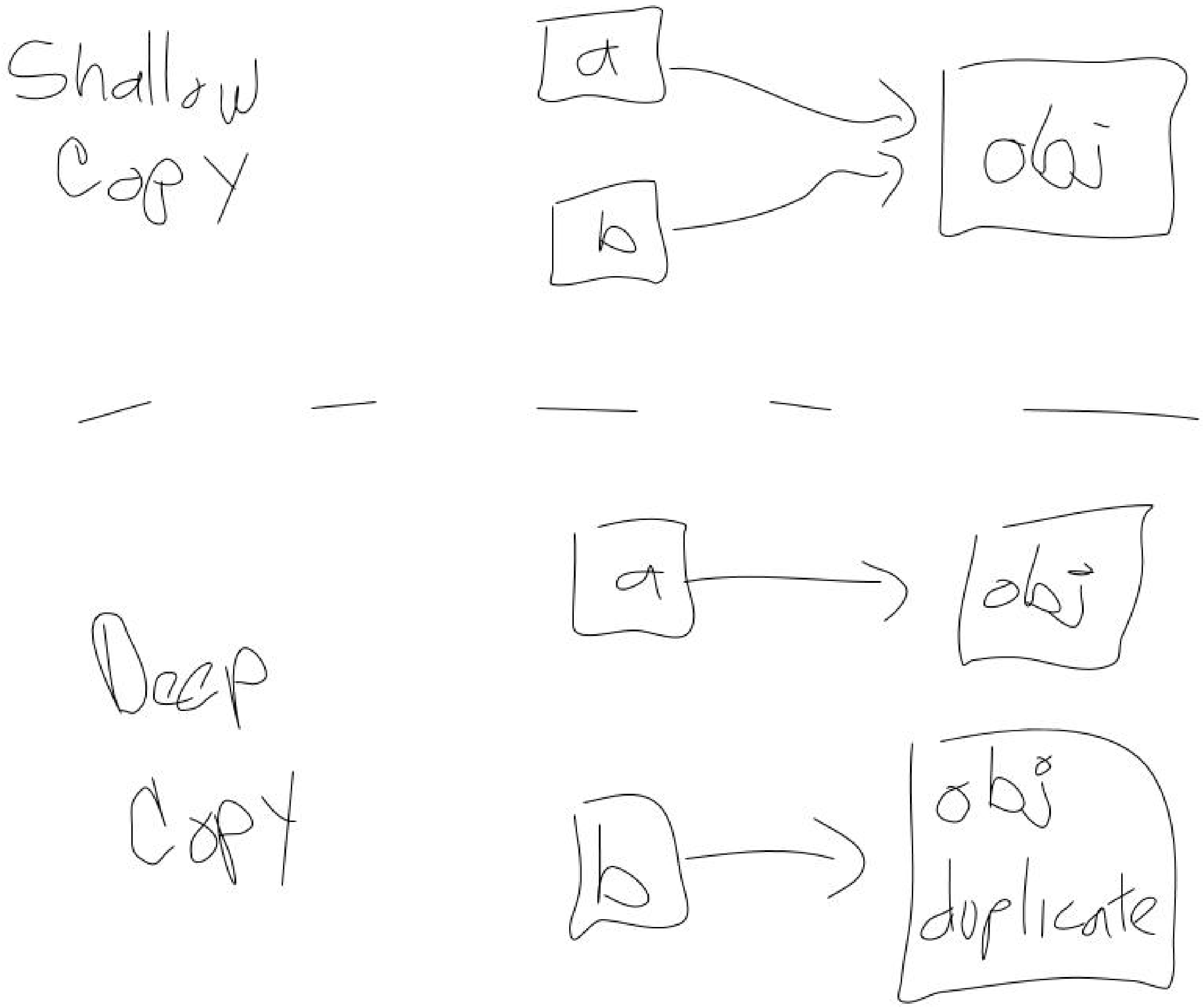
Memory Usage (cont): Deep vs Shallow Copy

- Data is copied often in computer programs
  - Assignment
  - Parameter passing
  - Return values
- Shallow copy – a simple bitwise copy.
  - May (or may not) be good when dealing with pointers
  - EG: pass by reference
- Deep copy – pointers are dereferenced and the data pointed to by the pointer is copied.
  - EG (C++): overloading the =operator or copy constructor

Copy Example

- Assume a and b are pointers. Assign b to a.

  a := b



Data Storage Hierarchy (in practice)

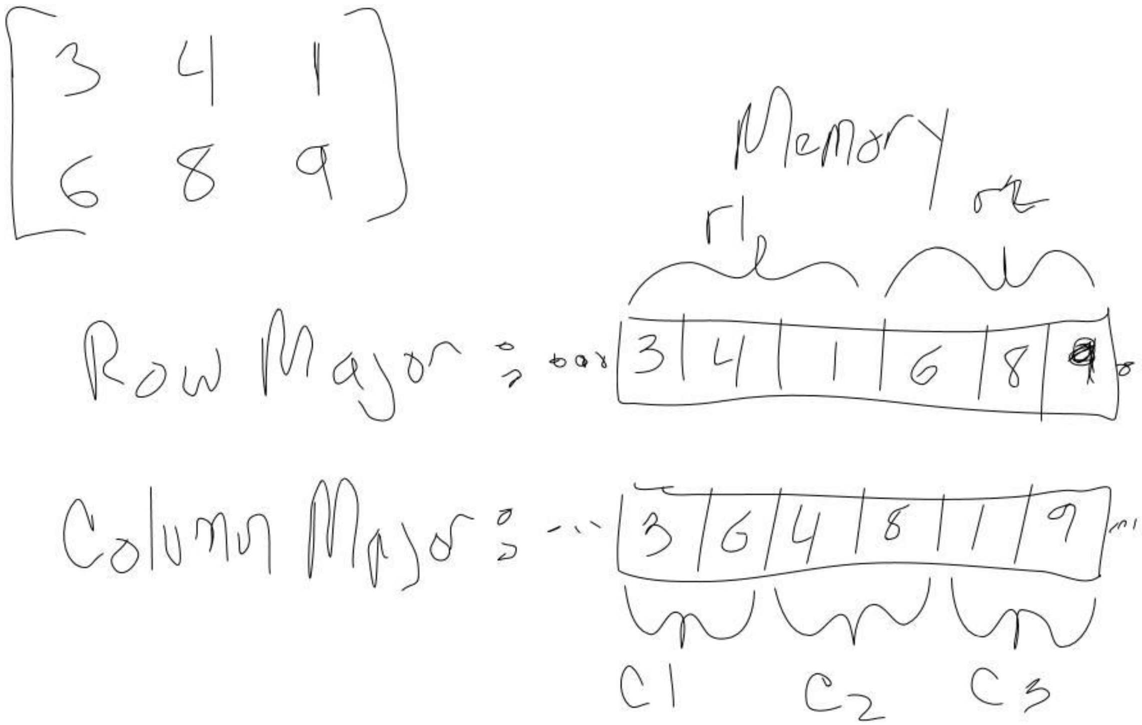
- Another practical issue related to algorithm analysis and implementation is RAM limitations and retrieving data from secondary storage.
  - Over-Simplified Summary: Retrieving data from secondary storage is *very slow*, and thus data is loaded to cache and RAM memory for processing. RAM is very versatile but has limitations. Some data structures may be large enough such that data will need to be continuously retrieved and restored back to secondary storage (thrashing).
  - Understanding RAM and Caching limitations and schemes on a computer is important when designing large data structures.

- Possible ways to mitigate these issues
  - Localize processing to subsets of the data.
  - Traverse and organize data smartly
    - Row-major vs column major

Code Example: Row Major vs Col. Major

```
// Efficient Traversal for Row -Major

for (int i = 0;i< numRows; i++)
        for (int j = 0; j < numCols; j++)
                c[i][j] = i + j;
```

```
// Efficient Traversal for Col-Major

for (int j = 0;j< numCols; j++)
        for (int i = 0; i < numRows; i++)
                c[i][j] = i + j;
```



Outline

I. Computational Complexity
   I. Time
      I. Upperbounding Notation
      II. Formalizing using Recurrences
   II. Space
      I. Recursion: Implications on Space Complexity

II. Memory Management and Implementation Details
   I. Pointers
   II. Allocation on Stack vs Heap

III. Good Programming Practices
   I. Design
   II. Debug

Good Structure Design Schemes

- Project design details are largely up to you, but I encourage you to abide by good coding practices.
  - This will benefit you and anyone who might use or read your code.

- Design for usability, reusability, efficiency, ….
  - Object Oriented Programming Design
    - Inheritance (writeability, reusability)
    - Encapsulation (reliability)
    - (Poly) Dynamic dispatch (writeability, flexibility, reusability)
  - Templates
    - Explicit “polymorphism” (reusability, writeability)

- Some simple computational complexity tips

OOP Design

- Classes / Structs provide for a fitting programming construct to represent data structures.

- Appropriately designing classes and class hierarchies will improve the effectiveness of your code
  - Reusability
  - Readability
  - Writeablility

Templates in C++

- Templates / Generics allow for explicit “polymorphism” which promotes
  - Flexibility
  - Reusability
  - Readability
  - Writeablility

Debugging and Testing

- Time distribution:
  - Planning Designing: 1/3
  - Actual Coding: 1/3
  - Testing / Debugging: 1/3

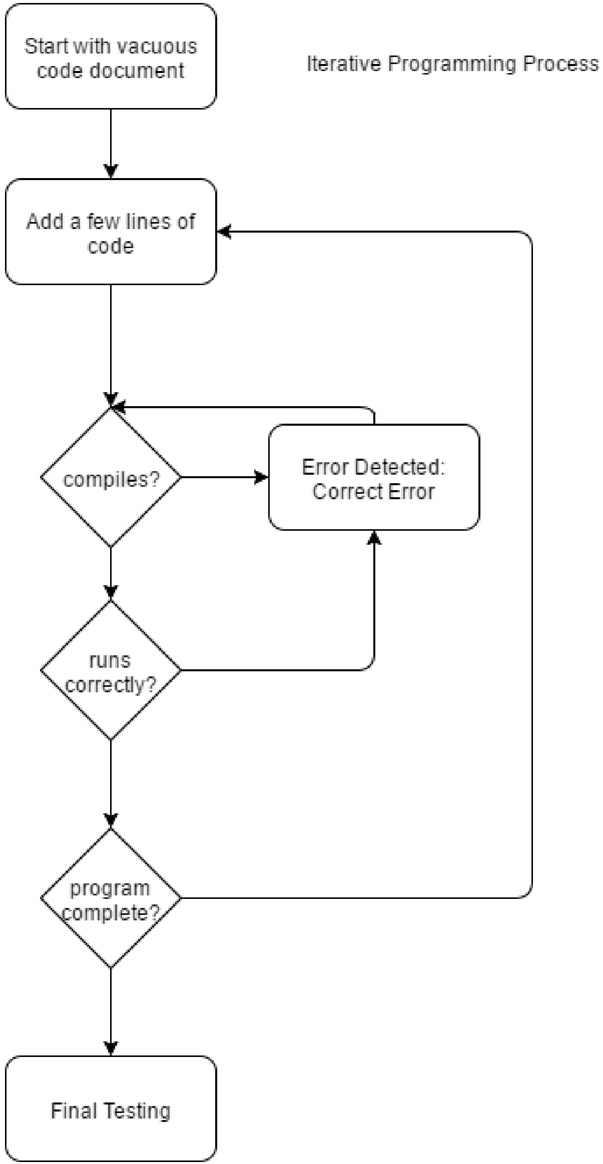
- Design a good testing scheme for your software.
  - Repeatedly code, then test, code, then test, …
  - In theory, you want to assure that your program produces the correct output for all possible inputs
  - In practice, select a subset of inputs that “efficiently” test your program.
    - EG: Include test input values such that each branch in execution flow is tested.

Programming Process (details)

1. Design
    1. Structural and procedural schematics
        1. UML class diagrams for all structures.
        2. Flow diagrams for all methods / procedures
    2. Confirm the correctness of your design with some examples.

2. Iteratively Code and Test
    1. Start with an empty code file
    2. Add one line of code (or a few lines of code): compile and if possible run on some sample input.
        1. If you have a compilation error or a runtime error, you have added a line of code that introduces an error (either syntax error or semantic error).
        2. The good news – you have found the error: it is on the line you have just added (almost always)! Finding the error is generally the hardest part of debugging – this is indeed good news!
        3. Fix the error and repeat.

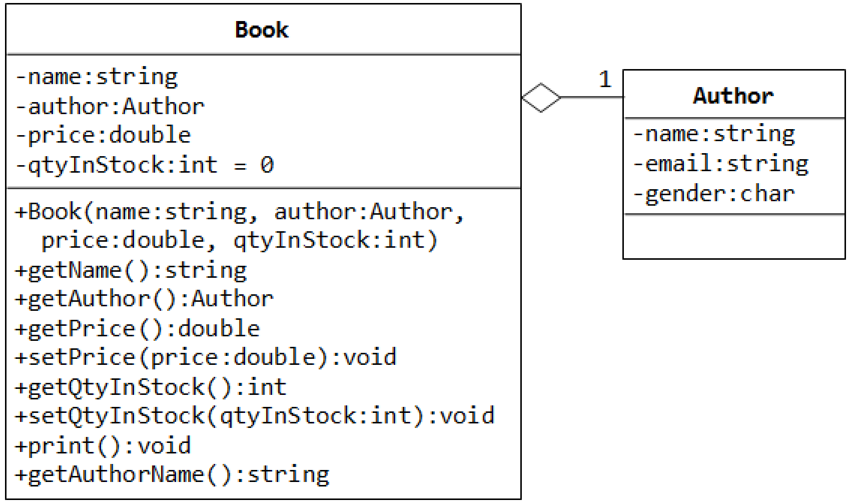
3. Final Testing
    1. Once your code seeming runs correctly on the sample input, try on a larger set of sample inputs. In theory, you want to assure that your program produces the correct output for all possible inputs – this is generally not practical, therefore you must create or select a set of inputs that test all or most potential flows of execution of your code.


Start with vacuous code document
Iterative Programming Process
Add a few lines of code
compiles?
Error Detected: Correct Error
runs correctly?
program complete?
Final Testing

Design

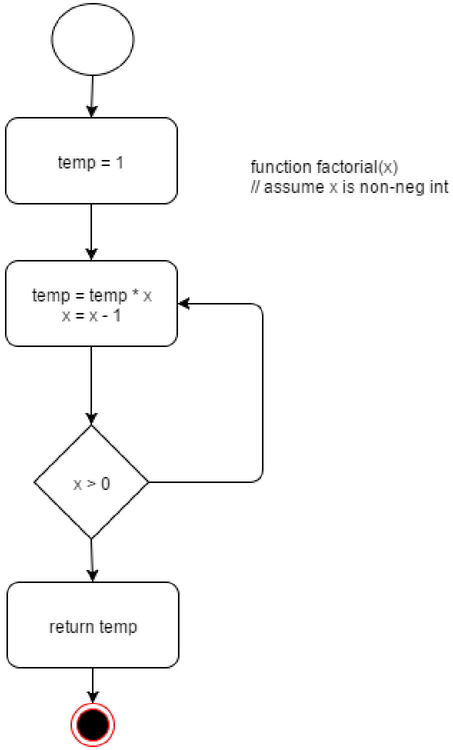
- Structural: entities or classes of your program
  - UML class diagram
  - ERD
  - Lists and may describe the attributes of the major components of your program (classes if using OOP)

- Procedural: the methods of your program
  - Flow diagram
  - Identifies the sequence of steps necessary to successfully complete a procedure or function.



```
Book
-name:string
-author:Author
-price:double
-qtyInStock:int = 0
+Book(name:string, author:Author,
  price:double, qtyInStock:int)
+getName():string
+getAuthor():Author
+getPrice():double
+setPrice(price:double):void
+getQtyInStock():int
+setQtyInStock(qtyInStock:int):void
+print():void
+getAuthorName():string
```

```
Author
-name:string
-email:string
-gender:char
```



function factorial(x)
// assume x is non-neg int

Some Final Computational Complexity Tips

- Time
  - If possible, remove statements from loops.
  - If possible, use operators that are fast.
  - Avoid excessive recursion.
  - Use mathematical properties to your advantage
  - Use practical understanding of computers to your advantage.

- Memory
  - Large structures:
    - Pass by reference / global*
    - Manage large structures “intelligently” to account for memory / hardware delays
  - Use “minimally sufficient” data types
    - EG: Don’t use a double if an int will suffice.
  - Avoid (excessive) recursion (unnecessary memory usage).
  - Use practical understanding of computers to your advantage.