# CS225L Lab 4: Algorithms, Pseudo-Code, and Computational Complexity

# Learning Outcomes

* Know what is an algorithm
* Get knowledge of pseudo-code and the difference between pseudo-code and programming code
* Learn how to compute the complexity of an algorithm
* Design two algorithms and analyze their complexities

# Pre-lab

**Algorithm Development**

An algorithm is a precise description of the steps that a computer needs to accomplish to solve a problem that is not tied to a specific programming language (Note: The currently fashionable term for not being tied to a specific language is language agnostic). Algorithms are written before any code is written. A complete algorithm provides proof that the problem can be solved by a computer, allows you to determine the amount of work you are requiring the computer to do (complexity), and serves as a model for the code to be written later.

A well written algorithm can be easily translated to nearly any programming language. In order to properly be classified as an algorithm, the following properties must apply:

1. Correct: The result must be 100% correct for 100% of possible inputs.

2. Discrete: Each step must have a definite beginning and end. No step starts until the previous step ends.

3. Finite: Each step is completed in a finite amount of time.

4. Unambiguous: Each step clearly provides one well-defined course of action.

5. Simple: Each step is achievable by standard processor operations (e.g. compare two numbers, add two numbers).

It is common for algorithms to be preceded by a short text description of what the algorithm is attempting to do. This allows the writer to de ne terms, data structures, and so on, so the algorithm can be understood. In your homework and project work, you should do exactly that: short description of a solution followed by the algorithm.

**Pseudo-Code**

As noted above, algorithms are not written in any specific programming language, but are written in a blend of code-like statements with a bit of natural language mixed in. There are wide variations in pseudo-code styles, which can confuse students when they first start writing their own algorithms. There are times when it is clear that the algorithm is written poorly. For instance, Example 1 below does not qualify as pseudo-code because it is simply a narrative description of a solution that does not show the properties listed above. Example 2 is simply Java code. Example 3 is acceptable, as it is not a narrative, provides detail as to each step the computer must accomplish, but is not restricted to a specific language.

**Example 1: Bubblesort Narrative**

Given an array of integers, start at the beginning of the array and compare the first two numbers. If the two numbers are in order, do nothing, otherwise swap their positions. Keep doing this until the array is sorted.

**Example 2: Bubblesort in Java**

public void bsort (int [] list) {

int n = list.length;

for (int i = 0; i < n; i++) {

for (int j = 0; j < n - 1; j++) {

if (list [i + 1] < list [i]) {

tmp = list [i];

list [i] = list [i + 1];

list [i + 1] = tmp;

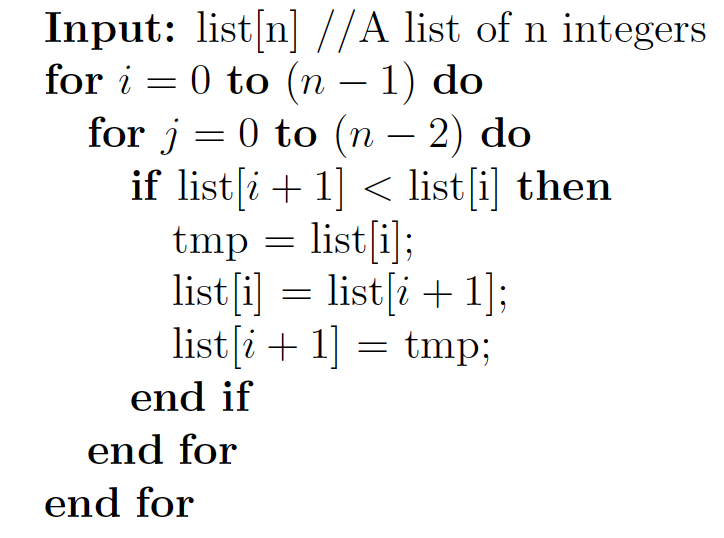
}

}

}

}

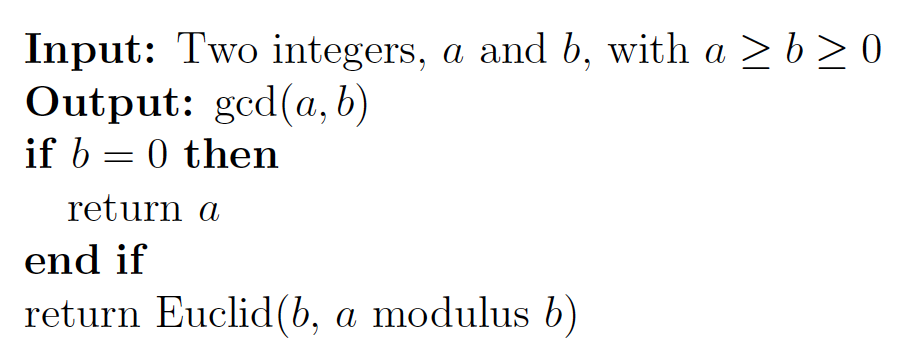
**Example 3: Bubblesort Pseudo-code**

****

Some forms use ‘fi’ and ‘od’ (reverse ‘if’, ‘do’) to close if and for loops, respectively.

Algorithms can be written quite tersely, but that may not be a good thing. Example 4 below is an algorithm for finding the greatest common denominator (GCD). It is taken from a book on algorithms, and it is completely correct. It is also di cult to understand and it may take you a few minutes (or a few days!) to convince yourself that it is correct. It is preceded in the text by about a page of explanation on Euclid's method for determining the GCD. For this class I recommend that you do not attempt to make your algorithms as short or compact as possible. Allow yourself a little more length for clarity.

**Example 4: Euclid (Source: Algorithms, by Dasgupta, Papadimitriou, and Vazirani)**



Further reading on algorithms:

1. As usual for computing and math topics, Wikipedia has a fairly good page on this. Of special interest is the fizzbuzz example in the Syntax section of the Pseudocode page.

See: <https://en.wikipedia.org/wiki/Algorithm>

Also see: <https://en.wikipedia.org/wiki/Pseudocode>

2. This example from University of Canterbury in Australia shows how pseudo-code is easier to understand than raw code.

See: [http://www.](http://www.cosc.canterbury.ac.nz/tim.bell/dt/Tutorial_Pseudocode.pdf) [cosc.canterbury.ac.nz/tim.bell/dt/Tutorial\_Pseudocode.pdf](http://www.cosc.canterbury.ac.nz/tim.bell/dt/Tutorial_Pseudocode.pdf)

3. This example from the University of North Florida shows, in my opinion, examples of how not to write pseudo-code. It does show how wide the ranging the term pseudo-code is. It also shows a writing style that you may use for early drafts of your pseudo-code while designing your algorithms.

See: <http://www.unf.edu/~broggio/cop2221/2221pseu.htm>

**Computational Complexity**

An algorithm is a precise description of the steps that a computer needs to accomplish to solve a problem that is not tied to a specific programming language. The completion of a correct algorithm is considered proof that software can be written to solve that problem. However, it is then important to consider how much work, or how many computer operations, is required to obtain the solution. The amount of work required by an algorithm is called its complexity. Software developers tend be concerned with average complexity (how much work the average problem takes to solve) and worse case complexity (the amount of time to solve given the worse case input). For this discussion, only worse case complexity is considered.

**Size of a Problem:** It may not be readily apparent, but specific in-stances of problems can be assigned a size, based on the amount of input given. For example, sorting is a well known problem that you have probably performed by hand, perhaps by putting a deck of cards in order. So would you rather sort a deck of five cards, or five-thousand cards? It is the same kind of problem (sorting), but the different sizes makes the amount of work very different. The critical question to be answered is “How much does the required work grow when the size of the input grows?” Algorithms with high computational complexity grow faster than those with low computational complexity. The variable n is used to designate the size of a problem.

**Big-Oh Notation:** The notation used to show an algorithm's worse case complexity is called Big-Oh notation, because a capital O is used in the notation. For example, one algorithm may have , or linear complexity, while another algorithm may have , or second order complexity. Standard complexities include: (constant cost), , , , , , and so on.

**Determining Complexity:** Determining the complexity of an algorithm is fairly straightforward process, once a few rules are understood:

Any portion of an algorithm that does not contain a loop and does not contain a function call is of constant cost:

Loops that iterate from 1 to , or 0 to (), are of linear order: .

If an algorithm is composed of two parts, with one being executed after the first is completed, the complexity of the entire algorithm is simply the larger of the two. Put another way, = the larger of the two.

If an algorithm is composed of two parts, with one nested inside the other, the complexity of the entire algorithm is equal to the individual

complexities multiplied together. Put another way,

Here are some examples using these rules. Assume algorithm A is , Algorithm B is and method foo() is .

If an algorithm executes algorithm A once followed by algorithm B once, the overall complexity is , since that is the larger of the two.

If an algorithm contains a loop that repeats times, and each time through the loop algorithm B is executed, then the overall complexity is (the complexity of repeating the loop) times the complexity of algorithm B: .

If an algorithm executes algorithm B twice in a row, the overall complexity is , which is the “larger” of and . Note: There is no such thing as , since coefficients are ignored.

If a loop that executes n times calls foo() once each time through the loop, the complexity is .

Loops nested two deep typically result in a complexity of , since each loop is , and . Similarly, loops nested three levels deep typically result in .

Algorithms with no loops and no function calls are , no matter how long they are.

**Problems and Complexity:** Good programmers are lazy. When asked to provide an algorithmic solution to a problem, you should not try to develop a new algorithm. You should try to find out if somebody else has already written an algorithmic solution, and almost always somebody has. Becoming familiar with typical problems is part of becoming a top-notch programmer. Below is a list of some well known problems and some of their complexities:

Searching (Unsorted List):

Searching (Sorted List):

Sorting:

Bubblesort:

Heap Sort:

Quicksort:

Mergesort: (Starting to think sorting is important?)

The Byzantine Generals Problem (also called the Two Generals Problem) Not solvable, depending on form.

The Convex Hull Problem Depth First Search

Breadth First Search Max-Flow:

Max-Cut:

The Graph Coloring Problem: (at best)

Knapsack Problem:

Maximal Clique Problem Minimum Cut Problem Minimum Spanning Tree

The Nearest Neighbor Problem (also called Nearest Neighbor Query):

SAT (also called Satisfiability):

Set Covering Problem

Traveling Salesman:

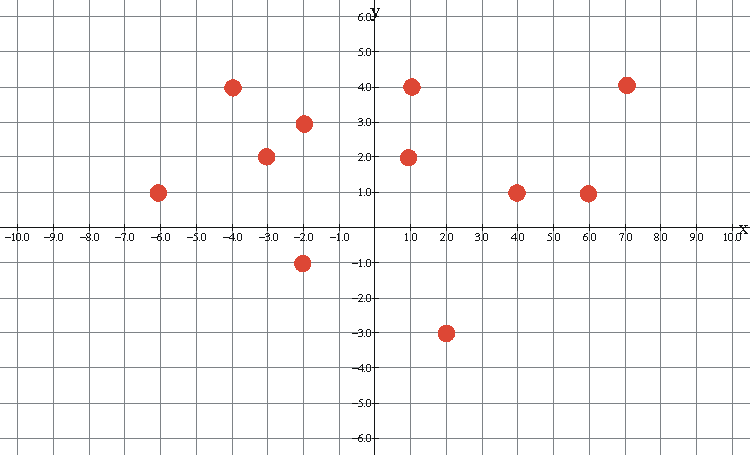
The Towers of Hanoi Problem

**Lab Activities**

In this lab you will be given two brief problem statements. Your will work in groups of three or four to develop an algorithm for each problem (10 points each). Once the algorithms are developed, also work to determine the worse case complexity of the algorithms you have developed (5 points each). Upload your algorithms and complexities in canvas.

**Problem 1: Bounding Box Problem**

Given a set of points find the bounding rectangle that just contains the points. The rectangle is aligned such that the edges are horizontal and vertical – there is no need to rotate the rectangle. (Hint: Basically you have to find the Xmax, Xmin, Ymax and Ymin of the points.)

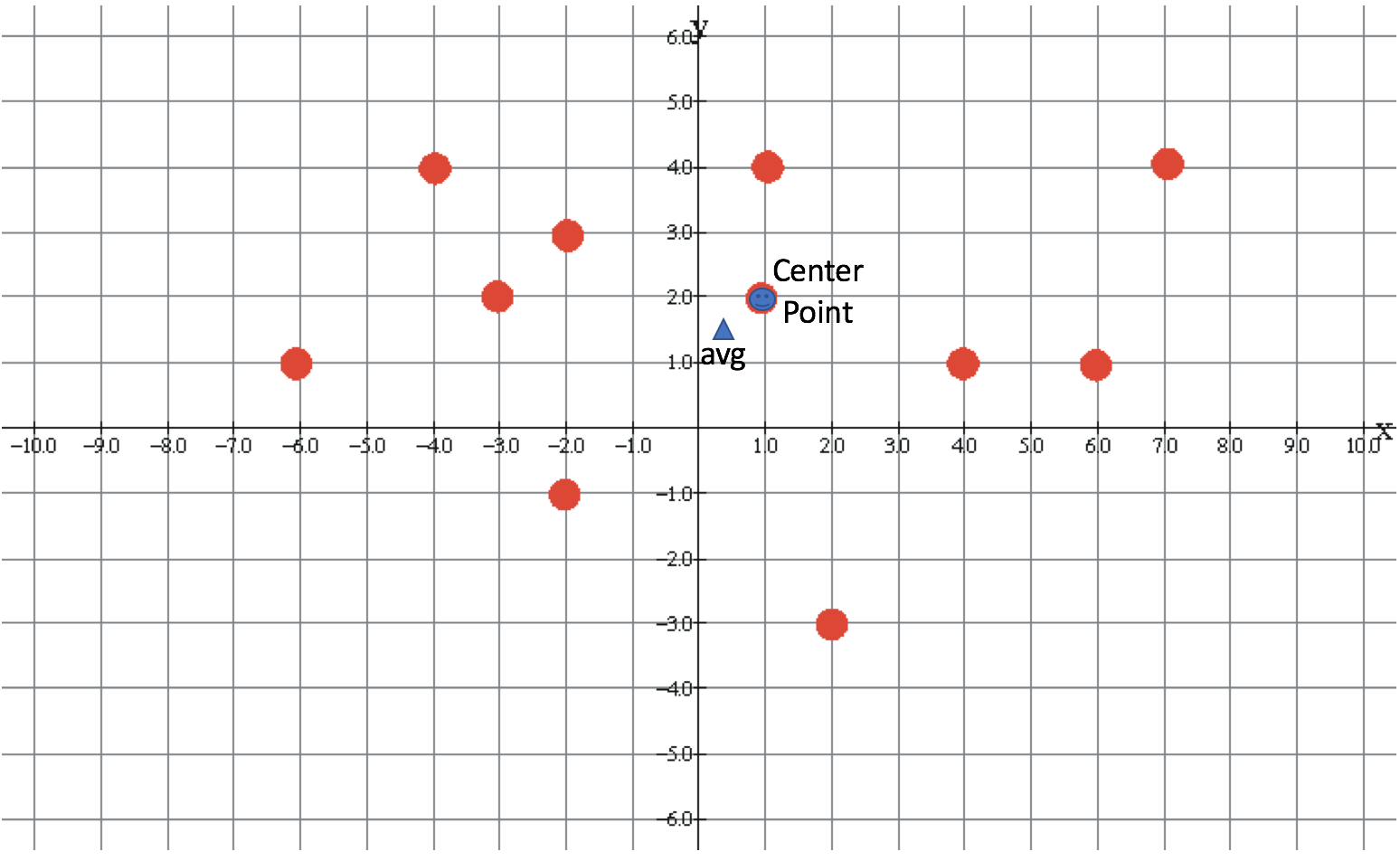


Input: a set of points

Output: an array includingthe top left point coordinates of the bounding rectangle , and the width and height of the bounding rectangle

**Problem 2: Center of Mass Problem**

Given a set of points in a X-Y plane, find the center point among the set of points. (Hint: Basically you need to find Xaverage and Yaverage of these points and then find a nearest point to (Xaverage, Yaverage))



Input: a set of points

Output: coordinates of the center point

**Deliverables:**

Please include your name, group member names, pseudo-codes and worst case complexities of each problem in your deliverables.