Final Project One

Dylan Coulter

CS300

**Vector pseudocode**

int numPrerequisiteCourses(Vector<Course> courses, Course c) {

totalPrerequisites = prerequisites of course c

**for** each prerequisite p in totalPrerequisites

**add** prerequisites of p to totalPrerequisites

**print** number of totalPrerequisites

}

void printSampleSchedule(Vector<Course> courses) {

**for** all courses

**print** out the course name

**if** the course has a prerequisite **Then**

**for** each prerequisite

**print** prerequisite course

}

void printCourseInformation(Vector<Course> courses, String courseNumber) {

**for** all courses

**if** the course is the same as courseNumber **Then**

**print** out the course information

**for** each prerequisite of the course

**print** the prerequisite course information

}

**Print Sorted List**

int partition(vector<Course>& courses, int begin, int end)

printSorted(courses)// Print method to print sorted courses

**Set** lowIndex as the first element

**Set** highIndex as the last element

**Set** midpoint equal to lowIndex + (highIndex-lowIndex)/2)

**Set** pivot equal to midpoint

**While** pivot is less than highIndex decrement highIndex

**Switch** the lower values to the left of the pivot

**Switch** the higher values to the right of the pivot

**Set** temp value equal to the lowIndex

**Set** the lowIndex equal to the highIndex

**Set** the highIndex equal to the temp

void quickSort(vector<Course>& courses, int begin, int end)//Creating a quicksort method

**Set** mid equal to 0

**Set** the lowIndex equal to begin

**Set** the highIndex equal to end

**If** begin is greater than or equal to end **Then**

**Return**

**Set** lowEndIndex equal to partition(course, lowIndex, highIndex)

**Init** recursive call to quicksort

**Init** quickSort(courses, lowIndex, lowEndIndex);

**Init** quickSort(courses, lowEndIndex + 1, highIndex)

void displayCourse(Course course)// created method to display courses

**Print** courseId “:” courseName “|” coursePrerequisite endline

**For** int i equals 0, i less than course.size(), ++i

displayCourse(courses[i])

void BinarySearchTree::inOrder(Node\* node) // creating inOrder method

**If** node is not equal to Null **Then**

**Init** inOrder node left// Check the left side first

**Print** courseId “:” courseName “|” coursePrerequisite endline

**Init** inOrder node right// Check the next leaf to right

**Print** courseId “:” courseName “|” coursePrerequisite endline

**Menu**

**Set** choice equal to 0

**While** user input not equal to 9

**Print** “Welcome to the course Planner”

**Print** “1. Load Data Structure.”

**Print** “2. Print Course List.”

**Print** “3. Print Course.”

**Print** “9. Exit”

**Print** “What would you like to do?”

**Create** switch(Choice)

**If** user input equals 1 **Then**

**Compute** Case 1: loadCourses(courseFile, Vector)

**If** user input equals 2 **Then**

**Compute** Case 2: printSorted(courses)

**If** user input equals 3 **Then**

**Compute** Case 3: printCourseInformation(courseId)

**If** user input equals 9 **Then**

**Compute** Case 4: End Program

**Else**

**Print** User Input “is not a valid option”

**END**

**Vector Run Time Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | 1 | 1 |
| **for each prerequisite p in totalPrerequisites** | 1 | n | n |
| **add prerequisites of p to totalPrerequisites** | 1 | n | n |
| **print number of totalPrerequisites** | 1 | 1 | 1 |
| **for all courses** | 1 | n | n |
| **print out the course name** | 1 | 1 | 1 |
| **if the course has a prerequisite Then** | 1 | n | n |
| **for each prerequisite** | 1 | n | n |
| **print prerequisite course** | 1 | 1 | 1 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Set lowIndex as the first element** | 1 | 1 | 1 |
| **Set highIndex as the last element** | 1 | 1 | 1 |
| **Set midpoint equal to lowIndex + (highIndex-lowIndex)/2)** | 1 | 1 | 1 |
| **Set pivot equal to midpoint** | 1 | 1 | 1 |
| **While pivot is less than highIndex decrement highIndex** | 1 | n | n |
| **Switch the lower values to the left of the pivot** | 1 | n | n |
| **Switch the higher values to the right of the pivot** | 1 | n | n |
| **Set temp value equal to the lowIndex** | 1 | 1 | 1 |
| **Set the lowIndex equal to the highIndex** | 1 | 1 | 1 |
| **Set mid equal to 0** | 1 | 1 | 1 |
| **Set the lowIndex equal to begin** | 1 | 1 | 1 |
| **Set the highIndex equal to end** | 1 | 1 | 1 |
| **If begin is greater than or equal to end Then** | 1 | 1 | 1 |
| **Return** | 1 | 1 | 1 |
| **Set lowEndIndex equal to partition(course, lowIndex, highIndex)** | 1 | 1 | 1 |
| **Init recursive call to quicksort** | 1 | 1 | 1 |
| **Init quickSort(courses, lowIndex, lowEndIndex);** | 1 | n | n |
| **Init quickSort(courses, lowEndIndex + 1, highIndex)** | 1 | n | n |
| **Print courseId “:” courseName “|” coursePrerequisite endline** | 1 | 1 | 1 |
| **For int i equals 0, i less than course.size(), ++i** | 1 | n | n |
| **If node is not equal to Null Then** | 1 | n | n |
| **Init inOrder node left// Check the left side first** | 1 | n | n |
| **Print courseId “:” courseName “|” coursePrerequisite endline** | 1 | 1 | 1 |
| **Init inOrder node right// Check the next leaf to right** | 1 | n | n |
| **Print courseId “:” courseName “|” coursePrerequisite endline** | 1 | 1 | 1 |
| **Total Cost** | | | 19n+36 |
| **Runtime** | | | O(n) |

**Hashtable pseudocode**

int numPrerequisiteCourses(Hashtable<Course> courses) {

totalPrerequisites = Hashtable c  
 **for** each of the prerequisite p in totalPrerequisites  
 **add** the prerequisites in Hashtable p to totalPrerequisites  
 **print** out the number of totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

**for** all key and value pairs in courses  
 **print** out the key course name  
 **if** the value has a prerequisites **Then**  
 **for** each prerequisites  
 **print** out the prerequisites

}

void printCourseInformation(Hashtable<Course> courses, String courseNumber) {

**for** all courses  
 **if** the course is equal to courseNumber **Then**  
 **print** out the course information  
 **for** each prerequisite of Hashtable[course]  
 **print** out the prerequisite course information

}

**Print Sorted List**

int partition(vector<Course>& courses, int begin, int end)

printSorted(courses)// Print method to print sorted courses

**Set** lowIndex as the first element

**Set** highIndex as the last element

**Set** midpoint equal to lowIndex + (highIndex-lowIndex)/2)

**Set** pivot equal to midpoint

**While** pivot is less than highIndex decrement highIndex

**Switch** the lower values to the left of the pivot

**Switch** the higher values to the right of the pivot

**Set** temp value equal to the lowIndex

**Set** the lowIndex equal to the highIndex

**Set** the highIndex equal to the temp

void quickSort(vector<Course>& courses, int begin, int end)//Creating a quicksort method

**Set** mid equal to 0

**Set** the lowIndex equal to begin

**Set** the highIndex equal to end

**If** begin is greater than or equal to end **Then**

**Return**

**Set** lowEndIndex equal to partition(course, lowIndex, highIndex)

**Init** recursive call to quicksort

**Init** quickSort(courses, lowIndex, lowEndIndex);

**Init** quickSort(courses, lowEndIndex + 1, highIndex)

void displayCourse(Course course)// created method to display courses

**Print** courseId “:” courseName “|” coursePrerequisite endline

**For** int i equals 0, i less than course.size(), ++i

displayCourse(courses[i])

void BinarySearchTree::inOrder(Node\* node) // creating inOrder method

**If** node is not equal to Null **Then**

**Init** inOrder node left// Check the left side first

**Print** courseId “:” courseName “|” coursePrerequisite endline

**Init** inOrder node right// Check the next leaf to right

**Print** courseId “:” courseName “|” coursePrerequisite endline

**Menu**

**Set** choice equal to 0

**While** user input not equal to 4

**Print** “Welcome to the course Planner”

**Print** “1. Load Data Structure.”

**Print** “2. Print Course List.”

**Print** “3. Print Course.”

**Print** “9. Exit”

**Print** “What would you like to do?”

**Create** switch(Choice)

**If** user input equals 1 **Then**

**Compute** Case 1: loadCourses(courseFile, Hashtable)

**If** user input equals 2 **Then**

**Compute** Case 2: printSorted(courses)

**If** user input equals 3 **Then**

**Compute** Case 3: printCourseInformation(courseId)

**If** user input equals 9 **Then**

**Compute** Case 4: End Program

**Else**

**Print** User Input “is not a valid option”

**END**

**Hashtable Run Time Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **totalPrerequisites = Hashtable c** | 1 | 1 | 1 |
| **for each of the prerequisite p in totalPrerequisites** | 1 | n | n |
| **add the prerequisites in Hashtable p to totalPrerequisites** | 1 | 1 | 1 |
| **print out the number of totalPrerequisites** | 1 | 1 | 1 |
| **for all key and value pairs in courses** | 1 | n | n |
| **print out the key course name** | 1 | 1 | 1 |
| **if the value has a prerequisites Then** | 1 | n | n |
| **for each prerequisites** | 1 | n | n |
| **print out the prerequisites** | 1 | 1 | 1 |
| **for all courses** | 1 | n | n |
| **if the course is equal to courseNumber Then** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of Hashtable[course]** | 1 | n | n |
| **print out the prerequisite course information** | 1 | 1 | 1 |
| **Set lowIndex as the first element** | 1 | 1 | 1 |
| **Set highIndex as the last element** | 1 | 1 | 1 |
| **Set midpoint equal to lowIndex + (highIndex-lowIndex)/2)** | 1 | 1 | 1 |
| **Set pivot equal to midpoint** | 1 | 1 | 1 |
| **While pivot is less than highIndex decrement highIndex** | 1 | n | n |
| **Switch the lower values to the left of the pivot** | 1 | n | n |
| **Switch the higher values to the right of the pivot** | 1 | n | n |
| **Set temp value equal to the lowIndex** | 1 | 1 | 1 |
| **Set the lowIndex equal to the highIndex** | 1 | 1 | 1 |
| **Set mid equal to 0** | 1 | 1 | 1 |
| **Set the lowIndex equal to begin** | 1 | 1 | 1 |
| **Set the highIndex equal to end** | 1 | 1 | 1 |
| **If begin is greater than or equal to end Then** | 1 | n | n |
| **Return** | 1 | 1 | 1 |
| **Set lowEndIndex equal to partition(course, lowIndex, highIndex)** | 1 | 1 | 1 |
| **Init recursive call to quicksort** | 1 | 1 | 1 |
| **Init quickSort(courses, lowIndex, lowEndIndex);** | 1 | n | n |
| **Init quickSort(courses, lowEndIndex + 1, highIndex)** | 1 | n | n |
| **Print courseId “:” courseName “|” coursePrerequisite endline** | 1 | 1 | 1 |
| **For int i equals 0, i less than course.size(), ++i** | 1 | n | n |
| **If node is not equal to Null Then** | 1 | n | n |
| **Init inOrder node left// Check the left side first** | 1 | n | n |
| **Print courseId “:” courseName “|” coursePrerequisite endline** | 1 | 1 | 1 |
| **Init inOrder node right// Check the next leaf to right** | 1 | n | n |
| **Print courseId “:” courseName “|” coursePrerequisite endline** | 1 | 1 | 1 |
| **Total Cost** | | | 17n+22 |
| **Runtime** | | | O(n) |

**Tree pseudocode**

int numPrerequisiteCourses(Tree<Course> courses) {

totalPrerequisites = left child of Node c

totalPrerequisites = right child of Node c  
**for** each prerequisite p in totalPrerequisites  
 **add** left Node of node p to totalPrerequisites

**add** right Node of node p to totalPrerequisites  
 **print** out the total number of totalPrerequisites

}

void printSampleSchedule(Tree<Course> courses) {

**for** all Nodes as courses  
 **print** out the course information  
 **if** course has a child node to the left **Then**  
 **print** out the left child node as the prerequisite  
 **if** course has a child node to the right **Then**  
 **print** out the right child node as the prerequisite

}

void printCourseInformation(Tree<Course> courses, String courseNumber) {

**for** all Nodes  
 **if** the course is the same as courseNumber **Then**  
 **print** out the node information  
 **if** course has a left child node **Then**  
 **print** out the left child node as prerequisite course information  
 **if** course has a right child node **Then**  
 **print** out the right child node as prerequisite course information

**end** the Function

**else**

**if** course has a left child node **Then**  
 **go to** left child Node  
 **if** course has a right child node **Then**  
 **go to** right child node

}

**Print Sorted List**

int partition(vector<Course>& courses, int begin, int end)

printSorted(courses)// Print method to print sorted courses

**Set** lowIndex as the first element

**Set** highIndex as the last element

**Set** midpoint equal to lowIndex + (highIndex-lowIndex)/2)

**Set** pivot equal to midpoint

**While** pivot is less than highIndex decrement highIndex

**Switch** the lower values to the left of the pivot

**Switch** the higher values to the right of the pivot

**Set** temp value equal to the lowIndex

**Set** the lowIndex equal to the highIndex

**Set** the highIndex equal to the temp

void quickSort(vector<Course>& courses, int begin, int end)//Creating a quicksort method

**Set** mid equal to 0

**Set** the lowIndex equal to begin

**Set** the highIndex equal to end

**If** begin is greater than or equal to end **Then**

**Return**

**Set** lowEndIndex equal to partition(course, lowIndex, highIndex)

**Init** recursive call to quicksort

**Init** quickSort(courses, lowIndex, lowEndIndex);

**Init** quickSort(courses, lowEndIndex + 1, highIndex)

void displayCourse(Course course)// created method to display courses

**Print** courseId “:” courseName “|” coursePrerequisite endline

**For** int i equals 0, i less than course.size(), ++i

displayCourse(courses[i])

void BinarySearchTree::inOrder(Node\* node) // creating inOrder method

**If** node is not equal to Null **Then**

**Init** inOrder node left// Check the left side first

**Print** courseId “:” courseName “|” coursePrerequisite endline

**Init** inOrder node right// Check the next leaf to right

**Print** courseId “:” courseName “|” coursePrerequisite endline

**Menu**

**Set** choice equal to 0

**While** user input not equal to 4

**Print** “Welcome to the course Planner”

**Print** “1. Load Data Structure.”

**Print** “2. Print Course List.”

**Print** “3. Print Course.”

**Print** “9. Exit”

**Print** “What would you like to do?”

**Create** switch(Choice)

**If** user input equals 1 **Then**

**Compute** Case 1: loadCourses(courseFile, Vector)

**If** user input equals 2 **Then**

**Compute** Case 2: printSorted(courses)

**If** user input equals 3 **Then**

**Compute** Case 3: printCourseInformation(courseId)

**If** user input equals 9 **Then**

**Compute** Case 4: End Program

**Else**

**Print** User Input “is not a valid option”

**END**

**Binary Tree Run Time Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **totalPrerequisites = left child of Node c** | 1 | 1 | 1 |
| **totalPrerequisites = right child of Node c** | 1 | 1 | 1 |
| **for each prerequisite p in totalPrerequisites** | 1 | n | n |
| **add left Node of node p to totalPrerequisites** | 1 | n | n |
| **add right Node of node p to totalPrerequisites** | 1 | n | n |
| **print out the total number of totalPrerequisites** | 1 | 1 | 1 |
| **for all Nodes as courses** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **if course has a child node to the left Then** | 1 | n | n |
| **print out the left child node as the prerequisite** | 1 | 1 | 1 |
| **if course has a child node to the right Then** | 1 | n | n |
| **print out the right child node as the prerequisite** | 1 | 1 | 1 |
| **for all Nodes** | 1 | n | n |
| **if the course is the same as courseNumber Then** | 1 | n | n |
| **print out the node information** | 1 | 1 | 1 |
| **if course has a left child node Then** | 1 | n | n |
| **print out the left child node as prerequisite course information** | 1 | 1 | 1 |
| **if course has a right child node Then** | 1 | n | n |
| **print out the right child node as prerequisite course information** | 1 | 1 | 1 |
| **end the Function** | 1 | 1 | 1 |
| **if course has a left child node Then** | 1 | n | n |
| **go to left child Node** | 1 | n | n |
| **if course has a right child node Then** | 1 | n | n |
| **go to right child node** | 1 | n | n |
| **Set lowIndex as the first element** | 1 | 1 | 1 |
| **Set highIndex as the last element** | 1 | 1 | 1 |
| **Set midpoint equal to lowIndex + (highIndex-lowIndex)/2)** | 1 | 1 | 1 |
| **Set pivot equal to midpoint** | 1 | 1 | 1 |
| **While pivot is less than highIndex decrement highIndex** | 1 | n | n |
| **Switch the lower values to the left of the pivot** | 1 | n | n |
| **Switch the higher values to the right of the pivot** | 1 | n | n |
| **Set temp value equal to the lowIndex** | 1 | n | n |
| **Set the lowIndex equal to the highIndex** | 1 | n | n |
| **Set mid equal to 0** | 1 | 1 | 1 |
| **Set the lowIndex equal to begin** | 1 | 1 | 1 |
| **Set the highIndex equal to end** | 1 | 1 | 1 |
| **If begin is greater than or equal to end Then** | 1 | n | n |
| **Return** | 1 | 1 | 1 |
| **Set lowEndIndex equal to partition(course, lowIndex, highIndex)** | 1 | 1 | 1 |
| **Init recursive call to quicksort** | 1 | 1 | 1 |
| **Init quickSort(courses, lowIndex, lowEndIndex);** | 1 | n | n |
| **Init quickSort(courses, lowEndIndex + 1, highIndex)** | 1 | n | n |
| **Print courseId “:” courseName “|” coursePrerequisite endline** | 1 | 1 | 1 |
| **For int i equals 0, i less than course.size(), ++i** | 1 | n | n |
| **If node is not equal to Null Then** | 1 | n | n |
| **Init inOrder node left// Check the left side first** | 1 | n | n |
| **Print courseId “:” courseName “|” coursePrerequisite endline** | 1 | 1 | 1 |
| **Init inOrder node right// Check the next leaf to right** | 1 | n | n |
| **Print courseId “:” courseName “|” coursePrerequisite endline** | 1 | 1 | 1 |
| **Total Cost** | | | 26n+23 |
| **Runtime** | | | O(n) |

**Evaluation**

The academic advisors in the Computer Science department at ABCU have asked us to create a program that will print a list of all the Computer Science programs in alphanumeric order. We were also asked to design the program to print out the course title and if the course required prerequisites to print those as well. We also will be asked to create a menu for user input to pull data on the course and display the results to the user. To complete this we create pseudocode for three different data structures that can be used to store and retrieve the data on the Computer Science programs at ABCU. The three data structures being evaluated are Vector, Hash table, and Binary Tree. After completing the pseudocode, we analyzed each data structure and calculated the big O notation for each to decide which was most efficient.

The first data structure we will be discussing is vector. A vector is a sequence container that represents arrays that can change in size during their runtime (Kim, 2020). When considering a vector data structure, we need to think about the pros and cons. A vector is very efficient when it comes to removing or adding elements. Vectors are dynamic in nature and multiple elements can be stored. When adding elements at the end of a vector this will happen in differential time. This is because the vector may need to be resized. When removing an element from the end of a vector this happens in constant time due to the fact that no resizing will have to take place. A con of a vector is adding or removing an element from the beginning or middle of a vector. Adding or removing from the beginning or middle of a vector happens in linear time and because of this would making a data structure that is not optimal for some tasks as the weakness is surrounding adding and removing elements from the beginning or middle (Great, 2022). Vectors do not use contiguous memory, are not indexed, and the memory consumption is greater than some other data structures.

The next data structure we will be reviewing to better understand the pros and cons of is the hash table data structure. Hash table is a form of data structure that uses the hash function to map keys to values (Ayushi, 2022). Hash tables store data in an array in an associative manner where each data value has its own special index because of this hash tables provide a secure way to retrieve and access data (Ayushi, 2022). One area for concern when considering hash tables is collisions that happen because the hash function gives each data value a special index and sometimes these indexes can be the same creating a collision. Collisions can be resolved using two different methods. First there is open addressing where you create a location for storing or searching the data called a probe (Ayushi, 2022). Second, we have closed addressing and this involves chaining where a linked list and an array are utilized. Some pros of utilizing a hash table are they provide better synchronization than compared to other data structures, hash tables provide constant time searching, and they have unlimited space as we can always add more elements to the chain. Some of the cons of utilizing a hash table are collisions as was discussed, hash tables do not allow for null values, keys are not stored in a special order making locating a particular key difficult. Also, many hash tables utilize linked list making them not cache-friendly as data is not put next to each other in memory (Gudabayev, 2021).

The last data structure we will be reviewing for overall effectiveness and the pros and cons is the Binary Tree data structure. A binary tree is a tree that consist of nodes with each node branching of two child nodes. Each node can have no more than two child nodes, one to the left and one to the right. There are four basic operations of a binary tree such as searching, insertion, deletion, and traversals (Ayushi, 2022). Searching the binary tree has six steps. First step is check if the tree is null if the tree is not null continue to the other steps. Step two compare the key to be search to the root of the binary tree. Step three if they key is lesser than the root search the left side of the subtree. Step four if the key is more than the root search the right side of the subtree. Step five if key is equal to root the search is successful (Ayushi, 2022). Step 6 repeat steps three, four, and five. When considering deletion within a binary tree there are three ways and this revolves around whether the node is a leaf node, has one children, or has two children (Ayushi, 2022). There are three ways to traverse a binary tree, and these are pre order which consist of traversing node, left, right. Then we have in order which will traverse the tree left, node, right. Last, we have post order which traverses through the tree left, right, node. When considering the binary tree some of the pros are when the tree is balance insertion and deletion is fast, they are extremely efficient, the code is easier to understand and follow than other data structures. Some cons of the binary search tree are a big focus on balance as unbalance tree will function poorly, accessing an element in a tree is slower than that of an array.

After working on the pseudocode for each data structure we turned our focus to calculating the big O for each. Big O notation allows programmers to analyze their code for time and space complexity of the code. Big O is important to choosing the correct data structure to use for the program for ABCU Computer Science team. Also learning about each data structure and researching the strengths and weaknesses also played a role in making the best decision on what data structure to choose. After careful review the data structure I will be choosing to utilize to create the final project for ABCU is the Binary tree Data structure. The big O result that was calculated for this data structure was O(n) and there are many benefits in choosing the binary tree data structure. First, when comparing the tree to other data structures such as the vector deleting and adding of elements can be done more efficiently than that of a vector where you are deleting or adding in the beginning, or the middle will slow the system. When comparing the Binary tree to a hash table data structure the worry is not that of where you add and delete but managing collisions which is one of the biggest areas of concern when utilizing a hash table. The main concern when working with a binary tree with be the balance of the tree as an unbalanced tree can create issues. Even with the concern of balancing the searching, inserting, traversing, deleting within the tree make it a good choice to utilize for the course data for the ABCU Computer Science program.

**References**

# Ayushi, A. (2022, June 07).  *Applications, Advantages and Disadvantages of Hash Data Structure* <https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-hash-data-structure/>

Ayushi, A. (2022, June 13). *Applications, Advantages and Disadvantages of Binary Search Tree*

<https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-binary-search-tree/>

Kim, J. (2020, August 07).  *The Basics of C++ Vector*

<https://velog.io/@jiselectric/The-Basics-of-C-Vector>

Great, L. (2022, July 04).  *Vectors in C++*

<https://www.mygreatlearning.com/blog/vectors-in-c/>

Gudabayev, T. (2021, May 09).  *Data Structures: Hash Tables*

<https://dev.to/tamerlang/data-structures-hash-tables-e1c>

**Resources**

Resource: <https://www.unf.edu/~broggio/cop3530/2220pseu.htm>

Resource: <https://www.techgeekbuzz.com/blog/how-to-write-pseudocode/>

Resource: <https://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html>