**Cal Poly Pomona**

**College of Engineering**

**Electrical and Computer Engineering Department**



ECE 480-01

Software Engineering

Group Project Report 3

By: Group 3

Nolan Chang

Rheza Budiono (Lead)

Bryan Chan

Jared Buo-en Huang

Mohammed Mirza

To: Dr. Tim Lin

May 22nd, 2018

# Table of Contents

[Table of Contents 2](#_Toc514679065)

[Overview 3](#_Toc514679066)

[Credit Distribution 3](#_Toc514679067)

[Part 1a – Errors and Problems in Software Explored 4](#_Toc514679068)

[Figure 1 – Floating-point arithmetic C++ code implementation in Visual Studio 2015 4](file:///C:\Users\phant\Documents\Class%20Materials\ECE%20480\Own%20Work\ECE%20480%20Group%203%20Report%203.docx#_Toc514679069)

[Figure 2 – Output of the C++ code implementation 4](file:///C:\Users\phant\Documents\Class%20Materials\ECE%20480\Own%20Work\ECE%20480%20Group%203%20Report%203.docx#_Toc514679070)

[Figure 3 – Octal calculation Python code using Python 2 IDLE 4](file:///C:\Users\phant\Documents\Class%20Materials\ECE%20480\Own%20Work\ECE%20480%20Group%203%20Report%203.docx#_Toc514679071)

[Part 1b – Errors and Problems in Software Tested 5](#_Toc514679072)

[Figure 4 – C++ Code 5](#_Toc514679073)

[Figure 5 – C++ Output 5](#_Toc514679074)

[Part 2a – ECE 304 Concepts 6](#_Toc514679075)

[Part 2b – ECE 304 Code Analysis 7](#_Toc514679076)

[Appendix A – Heapsort Code 9](#_Toc514679077)

[MohammedMirza\_ECE304\_Project5.cpp 9](#_Toc514679078)

[Node.h 9](#_Toc514679079)

[Node.cpp 10](#_Toc514679080)

[Heap.h 10](#_Toc514679081)

[Heap.cpp 10](#_Toc514679082)

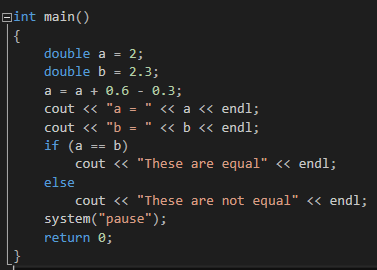
# Overview

Part 1a will explore computational errors involving different applications including floating point calculation errors in Visual Studio C++, the floor() function in MATLAB, as well as computations with leading zeros in Python 2.7.2. Part 1b will observe and compare the computational errors and differences with using different IDEs and languages to run the same computation. Part 2a will discuss programming concepts, specifically data structures – linked lists, stacks, and queues. Finally, part 2b will explain and elaborate on the functions of heapsort code developed for ECE 304 – Data Structures class.

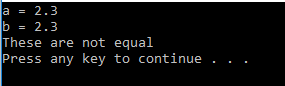
# Credit Distribution

Jared Huang wrote part 1a. Rheza Budiono completed the testing and analysis for part 1b. Nolan Chang wrote part 2a. Mohamed Mirza wrote and analyzed part 2b. And Bryan Chan did final editing, compilation, and formatting of the report.

# Part 1a – Errors and Problems in Software Explored

When it comes to calculation errors, there are many possibilities they can be attributed to. For example, there can be unique errors, such as compiler-specific and environment-specific. There can also be computational errors that can be replicated in all environments due to certain data structure properties or principles.

## **Figure 1** – *Floating-point arithmetic C++ code implementation in Visual Studio 2015*

 For instance, floating point arithmetic errors are on almost every platform. To assign a floating-point value, a computer must pack infinite real integers into a data structure to get as close as possible to the desired value. Since data structures are normally limited in size (whether it be 32-bit or 64-bit and onward), the infinite number of real integers must be represented by an approximation so that a finite value can be retrieved. Unfortunately, this approximation leads to a rounding error, and can result in inaccuracies when compared to a float that has not been applied an arithmetic operation. For example, in Microsoft Visual Studio, compiling and running the code shown in Figure 1 results in a floating-point precision error. The calculation results in an approximation that does not quite match the exact value of ‘2.3’. Hence, the output in Figure 2 is displayed. Therefore, when dealing with floating point precision, it is best to not compare exact values, but a range of values instead. In this way, the error will be compensated for and the correct result will be recorded.

## **Figure 2** – *Output of the C++ code implementation*

 This error can be seen even more obviously in MATLAB. Specifically, the floor() function rounds the parameter inside to the nearest integer less than or equal to itself. Simple code can be run from the command window, such as “floor(2.3\*50)”. Calculating this by hand results in a value of ’115’. In fact, calculation of this without the floor function in MATLAB results in an output of ‘115’ as well. Once the floor function is applied, the returned result is ‘114’. This is a slightly better representation on how the floating-point does not necessarily reach the theoretical value and is, in fact, a close approximation.

## **Figure 3** – *Octal calculation Python code using Python 2 IDLE*

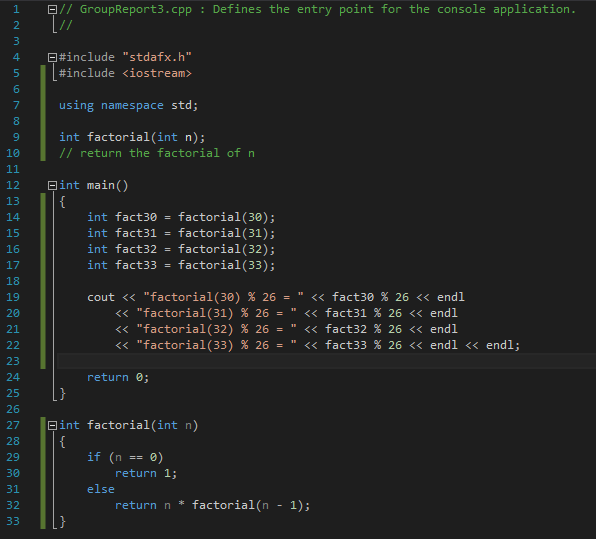
Another possible error stems from the inclusion of leading zeros in variable assignment. As of Python 2.7.2, the code shown in Figure 3 calculates and prints the integer result ‘17’. Running the same code in a program like MATLAB results in the computationally correct answer, ‘21’. This phenomenon must do with compiler-specific syntax. Most of the time, people who include leading zeros (whether by preference or organizational purposes) do so with the intention of calculating in base 10 (decimal). Unfortunately, the Python 2 IDLE reads it as a number in base 8 (octal). Therefore, the code does calculate the correct value, ‘21’, but in the wrong base. This sort of unspoken syntax specification has the potential to wreak havoc on code written by an unsuspecting developer. As a result, the assignment of a variable with leading zeros is disabled in Python 3, and only results in a syntax error.

# Part 1b – Errors and Problems in Software Tested

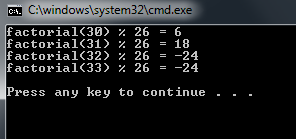
Upon trying on some different programming languages to see if they give the same errors, the results are quite surprising. We implemented the factorials of 31, 32, and 33 on C++, Python, and Excel. The only language that gives us the same mistakes as MATLAB is C++. As we can see on the C++ output figure shown above, it shows that the factorials of 30, 31, 32, 33 mod 26 are 6, 18, -24, and -24 respectively. And this is wrong because mathematically, factorial of 13 mod 26 would be the first factorial mod 26 that yields 0 since we will have 1 \* 2 \* 3 \* 4 \* … \* 12 \* 13, and in this term, we have 2 \* 13 which results in 26. Therefore, n factorial for n ≥ 13 would be 0.

Excel, however, yields the right results to a certain extent. As we can see from the Excel output above, 13, 14, 15, 16 factorial mod 26 yields 0. However, as the number gets larger, excel is giving an error “#NUM!”. I would assume that this was caused by a computational error, possibly because the result of the factorial is too large to be stored in the data type that excel provides. It may seem that the factorial result is fine, but we cannot be sure on that since the results are represented in scientific notation.

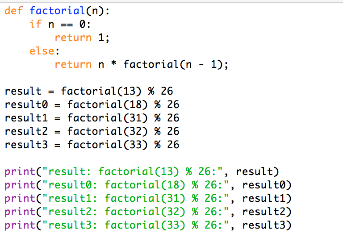
Finally, Python is the language that yields all right answer. I tested the factorial of 13, 18, 31, 32, and 33 mod 26, and these result in 0, which is the results that are supposed to be. One reason for why this works on Python would possibly be because we do not need to declare any data types on Python. Python could have assigned these numbers as Big Integer since Big Integer can store more number of digits as compared to a regular integer. This way, it allows the answers to be computed correctly.



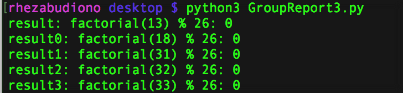
## **Figure 4** – *C++ Code*



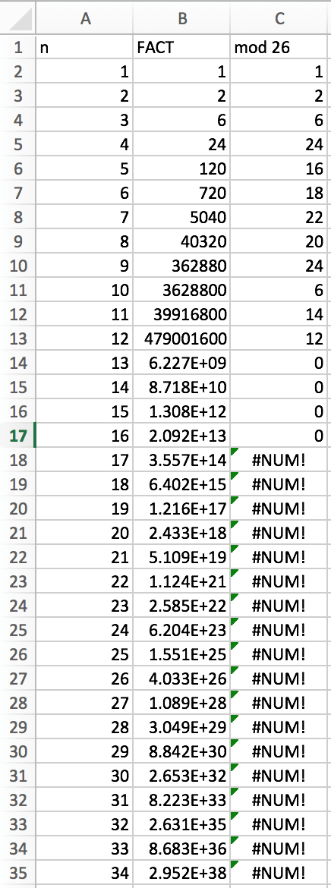
## **Figure 5** – *C++ Output*



**Figure 6** – *Python Code*



**Figure 7** – *Python Output*



**Figure 8** – *Excel Table*

# Part 2a – ECE 304 Concepts

ECE 304 covered data structures in programming and is the prerequisite of our current ECE 480 class. Throughout the course of the class, students learned about different ways to organize and store data. One of the first topics the class covered was linked lists, an enhanced version from a simple list. In a simple list, items are organized by a hierarchy defined by the programmer with the next item in the list directly following the previous item. A simple list can be implemented by using an array to store the items. However, one of the major drawbacks to the simple list is its linear characteristic and the fact that using arrays has its own problems. For one, when an item needs to be added to the simple list, the program needs to find the spot where the item is to be added and has to push back every single item after it. The same goes for deletion, but instead of pushing back, items need to be pushed forward to fill the space left by the deleted item. This leads to increased time for the program, and in the worst case, the entire list has to be pushed forward. The linked list solves this problem by using nodes to implement. In a linked list, nodes have a predecessor and a successor which can be switched easily. This solves the problem of deletion and insertion in the simple list, by improving the program time and making the program run more efficiently. Insertion and deletion in a linked list now only takes 1 time constant whereas in the simple list, insertion and deletion can take up to n time constants (n = total number of items in the list).

Another major topic covered in ECE 304 are queues and stacks, both of which are commonly seen in the outside world. A stack follows the rule of “last in, first out” which refers to the fact that the last item to go into the stack is the first to go out. Stacks use the term push when adding an element to the stack and pop when removing the most recently added item. For example, if I were to push “apple”, “pear”, and then “orange” to a stack, pop would return “orange”. Stacks are commonly found in computer memory, making it an important data structure. When implementing stacks using software, programmers can use either an array or nodes because there is no difference in program efficiency while using either one due to the nature of the stack. Queues are the opposite of stacks and are defined by its “first in, first out” nature. The queue term for adding an item is enqueue while dequeuer refers to removing an item. Using the example from before with the fruits, dequeue would return “apple” when using a queue. Queues are commonplace in every facet of daily life with the most common example being a line. The first person in a line is the first one that gets to get in or be serviced following the “first in, first out” behavior of a queue. Queues should be implemented in software with nodes rather than arrays because of the added time during deletion when using an array. When using nodes, simply snip the first node and return its value whenever a dequeue command is used.

# Part 2b – ECE 304 Code Analysis

***\*\*Note\*\* All code referenced in this section can be found in Appendix A***

In ECE-304, one of the projects that we had to do consisted of sorting a heap, a type of binary search tree where a node may have up to two children and one parent. The node with data of the highest value would be on the very top (in which case, the heap is a maximum heap) or on the very bottom (where the heap is a minimum heap). If a heap is a maximum heap, that means that every node’s parent has a value higher than the node itself. In a minimum heap, the opposite is true.

I was assigned the task of creating a list of random integers from 1 to 100 and storing them in a heap, which could’ve been either a minimum or a maximum heap. In my opinion, the maximum heap is simpler since searching for the highest value in the list would easily determine the head (the very top) node. Therefore, I decided to sort my list into a maximum heap. As can be seen below, I used a simple for loop to create one hundred random integers and store them into an array.

The second requirement for this project was to check whether the binary search tree was empty. An if statement coupled with the empty function are shown in the code. If the empty function returned a value of “true,” that would mean that the tree was empty. Otherwise, the tree had nodes in it. This was the easiest function for me to design because a tree has to have a head node, so I designed a function that let me check if there was a head node. If the tree did have a head node, we were assigned to show the tree in its entirety.

The third requirement was to design a function that would search for and return the largest data point from the heap. The largest value has to be the head node, which demonstrates the simplicity of the maximum heap. In contrast, if I had used a minimum heap, I would have to search through every node on the bottommost layer of the children using a for loop to determine which value was the highest.

The fourth requirement involved creating a remove function that would remove the head node (in this case, the largest value). While this may seem like a trivial task, it actually requires a lot of restructuring of the tree. For instance, the new head node (the larger of the two children) would have to be determined. Then, if that node already had two children and had to accommodate for a third (because of the second child of the original head node), the smallest node of the three would be the child of one of the other two. This had a trickle-down of complications that could only be resolved through using the insert function I created in the first part of the assignment.

The fifth and final requirement of the assignment ensured that the heap was properly sorted as a maximum heap (or a minimum heap, had that been my choice). As addressed in the previous requirement, the complications of the top node could shift the entire tree. A function had to be created to reorder the tree in a more manageable format. HeapSort made use of the function “retrieve,” so that the function knew where to start. Then, nodes would continue to be reassigned (i.e. the nodes the children of the head node) as long as the conditions of the two while loops were met. These two conditions checked whether there was a left child or a right child to every node, and if there was, to make a comparison between the two deciding which is the greater and if the parent is greater than the child.

While the requirements of the assignment may be easily understood, the design of the code is something I had trouble with for a long time for the duration of the assignment. A combination of .cpp and .h files have to be used for object-oriented programming in C++, since creating new classes in this language has its own complications. Thus, to create a node, a .cpp file and .h file with the node’s characteristics (e.g. the data in it, the node’s parent, etc.) have to be created. A heap is similar in that files of its own characteristics (which are similar to a typical binary search tree, with a head node and children nodes) must be developed to allow for the possibility of using such a data structure. The functions that were created and analyzed above were in the .cpp files, as they pertained to a specific task involving a heap.

The biggest issue I had when developing this code was the HeapSort function. While it didn’t take as many lines as some of the other functions, the logic behind how it worked versus how it was supposed to work took me many tests. I tested this function using “cout << “ and the name of the node I believed was not in the proper place. In the console output you may see below, the numbers aren’t perfectly sorted from greatest to least because a tree first outputs its head node, then its children (the smaller of which may be on the left, creating the illusion that the tree is not sorted properly). I had difficulty understanding this concept until I looked at the output for my first part, whereupon I realized the objective was not to create a list from greatest to least; it was to create a tree.

# Appendix A – Heapsort Code

What follows is Mohammed Mirza’s Code that was elaborated in part 2b. The code is broken up by files.

## MohammedMirza\_ECE304\_Project5.cpp

1. // MohammedMirza\_ECE304\_Project5.cpp : Defines the entry point for the console application. //
3. #include "stdafx.h"
4. #include "Node.h"
5. #include "Heap.h"
6. #include < iostream > using namespace std;
7. int main() {
8. cout << "Part 1: Insert" << endl << endl;
9. Node \* n1 = new Node(1);
10. Heap myTree(n1);
11. for (int i = 0; i < 99; i++) {
12. int temp = (rand() % 100) + 1;
13. n1 = new Node(temp);
14. myTree.insert(myTree.root, n1);
15. } //my insert function already sorts the data  //as you will see below //so heap sort may appear to change very little //depending on the random data //also, i am traversing the data using LVR //but the other two methods are available as well!
16. cout << endl << endl << "Part 2: Empty" << endl << endl;
17. cout << endl;
18. if (myTree.empty() == true) cout << "The list is empty!" << endl;
19. myTree.preorder(myTree.root);
20. cout << endl << endl << "Part 3: Retrieve" << endl << endl;
21. cout << "Here is the largest data: " << myTree.retrieve() - > data << endl;
22. cout << endl << endl << "Part 4: Remove" << endl << endl;
23. myTree.remove();
24. cout << "The first node is now gone!" << endl;
25. myTree.preorder(myTree.root);
26. cout << endl << endl << "Part 5: HeapSort" << endl << endl;
27. myTree.HeapSort();
28. cout << "The heap has been sorted!" << endl;
29. myTree.preorder(myTree.root);
30. cout << endl << endl;
31. return 0;
32. }

## Node.h

1. #pragma once
2. class Node {
3. public: int data;
4. Node \* right;
5. Node \* left;
6. Node \* parent;
7. Node(int data);
8. };

## Node.cpp

1. #include "stdafx.h"
2. #include "Node.h"
3. Node::Node(int d) {
4. data = d;
5. parent = '\0';
6. right = '\0';
7. left = '\0';
8. }

## Heap.h

1. #pragma once
2. #include "Node.h"
3. class Heap {
4. public: Node \* root;
5. Heap(Node \* );
6. bool empty();
7. bool search(Node \* root, int key);
8. void insert(Node \* root, Node \* n);
9. Node \* retrieve();
10. void remove();
11. void HeapSort();
12. void inorder(Node \* r);
13. void preorder(Node \* r);
14. void postorder(Node \* r);
15. };

## Heap.cpp

1. #include "stdafx.h"
2. #include "Node.h"
3. #include "Heap.h"
4. #include < iostream > using namespace std;
5. Heap::Heap(Node \* r) {
6. root = r;
7. }
8. bool Heap::empty() {
9. if (root == NULL) return true;
10. else return false;
11. }
12. bool Heap::search(Node \* root, int key) {
13. if (root == NULL) return false;
14. else {
15. if (root - > data == key) return true;
16. else {
17. if (root - > data > key) search(root - > left, key);
18. else search(root - > right, key);
19. }
20. }
21. }
22. void Heap::insert(Node \* r1, Node \* n) {
23. if (r1 == NULL) root = n;
24. else if (r1 - > data == n - > data) {
25. n - > data--;
26. insert(r1, n);
27. } else {
28. if (n - > data > r1 - > data) {
29. Node \* temp = r1;
30. root = n;
31. insert(root, temp);
32. } else {
33. if (r1 - > left == NULL) r1 - > left = n;
34. else if (r1 - > right == NULL) r1 - > right = n;
35. else {
36. if (n - > data >= r1 - > left - > data) {
37. Node \* temp = r1 - > left;
38. r1 - > left = n;
39. insert(r1 - > left, temp);
40. } else if (n - > data >= r1 - > right - > data) {
41. Node \* temp = r1 - > right;
42. r1 - > right = n;
43. insert(r1 - > right, temp);
44. } else {
45. Node \* succ\_left = r1 - > left;
46. Node \* succ\_right = r1 - > right;
47. int i = 0;
48. int j = 0;
49. while (succ\_left - > left != NULL) {
50. succ\_left = succ\_left - > left;
51. i++;
52. }
53. while (succ\_right - > right != NULL) {
54. succ\_right = succ\_right - > right;
55. j++;
56. }
57. if (j > i) insert(r1 - > right, n);
58. else insert(r1 - > left, n);
59. }
60. }
61. }
62. }
63. }
64. Node \* Heap::retrieve() {
65. Node \* largest = root;
66. return largest;
67. }
68. void Heap::remove() {
69. if ((root - > left != NULL) && (root - > right != NULL)) {
70. Node \* temp;
71. if (root - > left - > data > root - > right - > data) {
72. temp = root - > right;
73. root = root - > left;
74. } else {
75. temp = root - > left;
76. root = root - > right;
77. }
78. insert(root, temp);
79. } else if ((root - > left != NULL) && (root - > right == NULL)) {
80. root = root - > left;
81. } else if ((root - > left == NULL) && (root - > right != NULL)) {
82. root = root - > right;
83. } else;
84. }
85. void Heap::HeapSort() {
86. remove();
87. Node \* temp;
88. Node \* temp1 = retrieve();
89. Node \* temp2 = root - > left;
90. while (temp1 - > left != NULL) {
91. while (temp2 - > left != NULL) {
92. if (temp2 - > data < temp1 - > data) {
93. temp = temp2;
94. temp2 = temp1;
95. temp1 = temp;
96. }
97. temp2 = temp2 - > left;
98. }
99. temp1 = temp1 - > left;
100. }
101. }
102. void Heap::inorder(Node \* r) {
103. if (r != NULL) {
104. inorder(r - > left);
105. cout << r - > data << " ";
106. inorder(r - > right);
107. }
108. }
109. int x = 0;
110. void Heap::preorder(Node \* r) {
111. if (r != NULL) {
112. cout << r - > data << " ";
113. x++;
114. if ((x % 10) == 0) {
115. cout << endl;
116. }
117. preorder(r - > left);
118. preorder(r - > right);
119. }
120. }
121. void Heap::postorder(Node \* r) {
122. if (r != NULL) {
123. postorder(r - > left);
124. postorder(r - > right);
125. cout << r - > data << " ";
126. }
127. }