CSC 130

08/28/2018:

First Day of Class

* Coding for this class will be done in Java.
  + Should be fine, as I am already familiar with C++ and Java is C based
* Late submissions not accepted at all. No exceptions
* Only two exams throughout the course of the class: The midterm and the final
  + There may still be quizzes.
* Need to install eclipse on home desktop, possibly set up home desktop for remote access.
  + See syllabus for details on downloading eclipse dev environment
  + Will need to unzip a demo project (also downloaded from syllabus link) in order to start eclipse properly. Remember where I unzip it to as I will need to access it via its root URL.
  + Also need to edit properties for demo project and add the JRE system library, Junit 5, src/main/java, and src/test/java as code for running tests.
    - It’s similar to how I had to set up PyCharm. Worst case figure it out by trial and error.
* Assignments are graded based on ~4 tests which we are given per assignment + 1 test which the professor will add at whim, and will be scored both based on how many of the tests we pass (need to pass all 5 for max points) as well as the quality of the code which we write (such as run time analysis of the code itself).
  + When submitting assignments, only submit demoproject/src/main. Professor has the rest of the files already on her system.
* Look up Java review link, found in syllabus. Specifically look into auto-boxing/auto-unboxing
* This is going to be my hardest class.

08/30/2018:

* Analysis of algorithms
  + Time: How long does the program take run?
    - Run time complexity. Think O(n) complexity
  + Space: How much memory does it require?
* Factors
  + Input data, hardware platform, algorithm, language, other applications, etc.
* Model
  + Assumptions
    - Each simple operation (+, -, if, call, comparison, etc) takes one time unit
    - Each memory access takes one time unit
    - Infinite amount of memory, no cost associated with paging, swapping ,etc.
  + Problem size (input size)
  + Asymptotic efficiency
    - Problem size large enough
    - Only order of growth is relevant
  + Independent of hardware, language, other applications, etc.

Asymptotic Notations - O (Big Oh)

* f(n) = O(g(n)) if there are positive constants c and n­­0 such that f(n) ≤ cg(n) when n ≥ n0
* The growth rate of f(n) is less than or equal to that of g(n).
  + g(n) is an upper bound of f(n)
* You have bad runtime complexity if f(n) grows more quickly than g(n)

Asymptotic Notations - ω

* See lecture notes posted online on canvas. Will link HERE once posted

09/04/2018

Running Time Analysis

* Understand which case(s) is(are) asked for
  + Best-case, average-case, worst-case
    - Best-case time provides lower bound for all inputs
    - Average-case time presents typical behavior, in many cases average-case analysis is very complex and requires knowledge of distribution of all possible inputs.
    - Worst-case time provides a upper bound for all inputs
* Get running time function
  + Sum up all basic operations executed by an algorithm
  + Shortcuts can be taken to ignore insignificant parts
* Compute Big-Oh
  + A tight bound is expected in this class unless specified the other way
    - 2n2 + 5n = O(n2) is expected

09/06/2018:

* Space-complexity analysis
* See class notes for details.

09/11/2018:

**Recursion**

* A recursive method is one that calls itself
* Key elements of recursion
  + Best case(s)
  + Reduce larger problem into smaller one(s), which eventually is reduced to best case(s)
* Watch out for infinite recursion
* Recursion vs. iteration
  + Any recursive method can be implemented iteratively
  + Some problems are simpler and more natural if solved recursively
  + O(n) recursive calls use at least O(n) space (without tail recursion optimization)

**Practice**:

Use recursion and iteration to make a function which calulates N! and returns 1 when n = 0

int recurFact(int n)

if (n<= 0)

return 1

return n + recurFact(n-1)

void iterFact(int n)

int x = 1

while (n > 0)

x+= x\*n

n--

return x

09/13/2018:

Merge Sort:

* Why would I use this instead of an insertion sort?
  + Insertion sort has worse run time complexity
  + Merge sort runs in n\*log(n) time, while insertion runs in quadratic time (worst case).
  + Merge and Insertion sorts are both stable sorting algorithms

Exercise: Bottom down merge sort:

55| 61| 22| 97| 56| 96| 96| 88| 57| 49| 31| 48

55 61| 22 97| 56 96| 88 96| 49 57| 31 48

22 55 61 97| 56 88 96 96| 31 48 49 57

22 55 56 61 88 96 96 97| 31 48 49 57

22 31 48 49 55 56 57 61 88 96 96 97

09/18/2018:

Quick Sort:

* Best case: n\*log(n) run time complexity, log(n) space complexity
* Worst case: quadratic (n2) run time complexity, n space complexity
* Unstable sorting algorithm

Examples:

1. 1 2 3 4 5

1 | 2 3 4 5

1 | 2 | 3 4 5

1 | 2 | 3 | 4 5

1 | 2 | 3 | 4 | 5

1. 5 5 5 5 5

5 5 | 5 | 5 5

5 | 5 | 5 | 5 | 5

1. 5 4 3 2 1

1 4 3 2 | 5

1 2 3 | 4 | 5

1 2 | 3 | 4 | 5

1 | 2 | 4 | 4 | 5

10/04/2018:

class tree {

int rightHeight = 0;

int leftHeight = 0;

Node getTreeHeight(node){

if(node.right)

rightHeight = getTreeHeight(right);

if(node.left)

leftHeight = getTreeHeight(left);

if(leftHeight > rightHeight)

return leftHeight + 1;

return rightHeight + 1;

}

}

int main(){

int height = 0;

Node newTree = get(Node);

if (newTree == null){

height = -1;

}

else {

height = getTreeHeight(newTree);

}

return;  
}