Chapter I Terms and Concepts:

**Data Structures** – a nebulous, abstract concept that deals with how information is handled within a program. Often related to metaphors, they can be used to create more complex and productive programs without the requirements of redundancy. It is the corner stone of all advanced programming.

**Abstraction** – the process of dividing up a software program into component parts to maintain data integrity and manageability.

**Stack –** A collection of managed data, normally referred to as a first in last out system of organization. Consider it like a stack of dishes, where you have to remove the top ones first before you can get to the middle or bottom. Compare to Queue.

**Queue** – A collection of managed data which has a last in, first out sort of organization. When a queue is created, the bottom is pushed forward until it reaches the front of head, where it can then be accessed by the program. Queues can be programmed to ‘round-robin’, where things are moved around until the desired data is found and then reset.

**Information Hiding** – the ability to deal with abstract ideas, and are the primary tools used for advanced and complex systems.

**Software Development** – a team effort in most cases, this is a situation that requires teams of programmers, usually in the dozens. Information management is the largest obstacle to speedy and efficient development. Abstraction and information hiding are a key to success.

Chapter II – Concepts and Terms:

**Program** ­– A series of commands used to make a computer perform a useful or otherwise repetitive, mundane task. There are many forms of programs, but they all share one thing in common – they are frequently divided into smaller units and depending on your language of preference, are called functions (C/C++, C#, Javascript), procedures (Assembly, Java), methods (Python), subroutines, subprograms, classes, or packages.

**Function** – an easy way to think of a function is almost like a sub-program. It is performing a certain task based on the instructions input into the computer. A function in programming is a root form of abstraction.

**Abstraction** – (More In Depth): in the terms of functions, it is form hiding of information. It allows the user or programmer to skip over repeating lengthy instructions and reduces the amount of visible information from the outside. As programmers, we often use the same exact set of instructions multiple times throughout a program. As we do not need to know the internal details as users, this form of abstraction is also frequently called *Encapsulation.*

**Use (Function) –** generally the parameters or arguments for a function in the function call. Some functions operate without parameters. Deals with the input and output on a function. Use of a function requires only minimal knowledge. This is how an end user frequently uses a function.

**Write (Function)** – The set of instructions and creation of said instructions to be used inside of a function. Deals with how the inputs and outputs are handled. This requires intimate knowledge of the intent of a function.

**Algorithm**  - an algorithm is a description of how a specific problem will be solved, and written at a level of detail that can be followed by the reader, who then translates it into machine language or code. Terms that are commonly interchanged with algorithms are *process, routine, technique, procedure, pattern,* and *recipe.*

NOTE: It is critical to know the distinction between an algorithm and a function. Functions are a set of instructions, written in a programming language, where as an algorithm is simply a process that is written in to an understandable format. Basically, all functions are algorithms, but not all algorithms are functions. While functions change from language to language, algorithms stay fundamentally the same no matter what environment you place them in.

**Properties of the Algorithm:** *Every algorithm must possess the following qualities.*

**\* input preconditions:** these are the first steps in an algorithm. You create an argument/parameter set for the algorithm to evaluate. In most cases, there will be what is known as a base case, which is essential for recursive functions and algorithms, so that there is an ever increasing level of simplicity. Algorithms will evaluate the problem and inputs in relation to the base case and then create an output that will generally be revealed at the end of the algorithm.

\* **Precise Definitions of Each Instruction**: each step of the algorithm must be well defined. Unlike natural language, there can be no ambiguity about the actions to be carried out. Algorithms are presented in an informal descriptive form and are not generally well understood in natural languages without mathematical or other scientific nomenclature.

\* **correctness**: an algorithm is expected to solve a problem. It must follow through all arguments and logic to reach the desired solution. If the solution is incorrect, the algorithm and inputs must be re-evaluated for correctness. A fully formed algorithm is often called putative.

\* **termination, time to execute**: It must be clear that for any input values, the algorithm is guaranteed to terminate after a finite number of steps. Most algorithms have bounds, and cases which tell us how many steps it will generally take. It is often written as m + n. Time to execute is an abstract concept – which will be discussed more in depth in chapters 3 and 4.

**\* description of the result or effect**. Finally, it must be clear exactly what the algo is intended to accomplish. More often than not, this can be expresses as the resultant value. Sometimes there is something called a *side effect.* However, the outcome expected must be completely specified.

**Input Types** – Algorithms will in general produce only what is specified by the program. A data type contains certain forms of information based on the said data type. Most data types are numeric, but there is are two based on alphanumeric characters – char and string in most high level languages that support them. Many languages are now available that simply use on the fly typing, so specification is not always necessary. However, this can cause some algos to be unreliable or easily corrupted in programming languages that do not have string typing.

\*Note Zeno’s Paradox: An infinite algorithm. Read the course materials, which basically talk about an infinite case algorithm, given that things such as dividing up a candy bar will reach an ever smaller amount as steps are taken. This creates the need for base cases.

**Recursive Algorithms** – recursion is a common technique in algorithms AND life in general. Recursion at its most basic level is simple repetition of a function to create a desired result. Recursion is used frequently to ‘reduce’ the complexity of problems by ‘simplifying’ it in some way. This is often repeated until a thing known as a *base case* is reached. As you work through the algorithm, you may encounter what are known as *recursive* and *inductive cases.*

**Base Case –** the simplest, most fundamental, and otherwise reductive part of an algorithm or function. When this is reached, the algorithm is completed and stops, as it can no longer be simplified further.

**Recursive (Inductive) Cases** – these are steps that are not the base case and are used to prove and evaluate an algorithm or function.

**Activation Record Stack** – this is the ordered series of items in a stack. Previous values are piled up and gradually removed until the stack is totally cleared. The activation record stack simply refers to the stack itself. This is also known as the CALL STACK (see CS271 – Assembly)

**Towers of Hanoi –** a popular puzzle that is frequently used to demonstrate how recursions and algorithms work. The goal of the puzzle is to move all disks from pole A to pole B, using Pole C as a holder. However, no disk that is larger than another can be stacked upon a smaller disk. Also, you can only move one disk at a time.

Study Questions

1. **What is abstraction? Explain how a function is a form of abstraction.**

Abstraction is breaking programs into discrete parts to reduce redundancy and repetition. A function is a set of instructions that is then called by the user or program to perform some useful task. As abstraction is related to information hiding, the function is a reduced data form and reduces overall programmatic complexity.

1. **What is an algorithm?**

An algorithm is a set of instructions that remains fairly immutable across various languages, both programmatic and natural. It is used as a way to solve a problem. Mathematical formulas are a form of algorithm, as are all forms of functions.

1. **What is the relationship between an algorithm and a function?**

A function is the translation of an algorithm into a programming language. A function is used to process the algorithm in the computing environment.

1. **Give an example input condition that cannot be specified using only a type.**

A range restriction. Most algorithmic functions will produce an error or exception if an erroneous input is placed in.

**5. What are two different techniques used to specify the input conditions for an**

**algorithm?**

Use of type and type signature? (Range restriction is not necessarily allowed)

**6. What are some ways used to describe the outcome, or result, of executing an**

**algorithm?**

Based on the initial conditions, we can find the result type of an algorithm, which is defined as part of the function signature. There is also documentation that comes along with the result and relationship of the inputs.

**7. In what way does the precision of instructions needed to convey an algorithm to**

**another human being differ from that needed to convey an algorithm to a computer?**

Algorithms must be specified in a way that can result in a desirable outcome. Psuedo-code is often a good way to emphasize the essential aspects of the algorithm. Also, a computer needs you to be specific to the smallest detail whereas a human can give a little leeway and self-correction.

**8. In considering the execution time of algorithms, what are the two general types of**

**questions one can ask?**

How many times will the steps within an algorithm execute and what will ensure the algorithm executes properly.

**9. What are some situations in which termination of an algorithm would not be**

**immediately obvious?**

If there is no definite limit, it may not be obvious that the algorithm terminated.

**10. What are the three properties a value must possess in order to be used to prove**

**termination of an algorithm?**

1. the property or value can be placed into a correspondence with integer values.
2. The property is nonnegative
3. The property or value decreases steadily as the algorithm executes.

**11. What is a recursive algorithm? What are the two sections of a recursive algorithm?**

Recursion is a technique used to invoke and reduce the complexity of a problem, by ‘simplifying’ in some way. It is repeated until a base case is met. There are two special cases for recusion and they are called the base (most simple) case and the recursive or inductive steps/cases.

**12. What is the activation record stack? What values are stored on the activation record**

**stack? How does this stack simplify the execution of recursive functions?**

It’s part of the recursion process, and they are the values that are hidden/stored in the computer memory. The objects/memory stored in the activation stack are generally the condition arguments.

Chapter 3 – Terms and Concepts – Debugging and Correctness

**Debugging**  - a concept we have run into frequently up to this point. This allows a programmer to find, correct, and remove bugs and errors, as well as create a reliable and error-free (relatively) program. It’s testing for errors.

General Tips for Debugging: Test small portions of the program in isolation, see if errors are produced for given inputs and to see if error is consistent. If error is due to incorrect input handling, play with the problem and find the logical error. Think about what happens prior to the error. Use breakpoints and statements to isolate the bug. Don’t assume anything. Use assertions and invariants to reduce problem areas, and again, never make assumptions.

**Assertions and Invariants –** Assertions are a powerful tool, which is essentially a comment that stands in at a certain point of the program and evaluate the flow of the program and control it. A common form of this are if-else blocks. Asserts can also be invoked with a special source/header file assert.h in C.

Invariants are assertions within a loop, ensuring the right thing happens inside of the loop – or at least asks to make sure the right thing happens in the loop.

**Assertion Statement –** performs a task that is similar but not exactly the same as the concept of the assertion described above. The assertion statement, on the other hand, takes as argument an expression, and typically will halt execution and print an error message if the statement is not true. See above regarding assert.h. Example statement from reading:

double sqrt (double val)

{ assert (val >= 0); /\* halt execution if value is not legal \*/

**Bubble Sort** – the most inefficient sort but common in teaching computer science to help students understand the basics of sorting and search. It is extremely simple and easy to understand.

void bubbleSort (double data [ ], int n)

{

for (int i = n-1; i > 0; i--) {

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

// data[j] is largest value in 0 .. j

if (data[j] > data[j+1]) swap(data, j, j+1)

// data[j+1] is largest value in 0 .. j+1 } data[i] is largest value in 0 .. i }

// array is sorted }

**Binary Search** – binary search is based on the idea of dividing search areas into halves, until we find the result. It works by going for the center of the array and dividing it into composite parts and searching within those part until the desired information is found.

**Testing and Boundary Cases** – one should always use testing to figure out if a program is working correctly. And individual function testing is called unit testing. When you are looking to see what the limits of a function or data are, then try what is called boundary testing. This will help you isolate problems in the code or logical inconsistencies. The boundary testing is to see if something evaluates either true or false under given conditions, and where those limits are.

**Black Box Testing** - testing that considers only the structure of the input and output values, and ignores the algorithm used to produce the result.

**Integration Testing –** done after individual functions are tested, and are then tested together.

**Regression Testing –** going back and isolating any errors that changes may have introduced.

**White Box Texting** – Testing that considers every aspect of the program and function, finding out what makes something true, and what makes something false.

1. What does it mean to say you are debugging a function?

Debugging a function is going through a function and looking for errors in the code or problems with the logic and identifying them, removing them, and testing them to improve the functionality of the program. It is one of the biggest parts of systematic testing.

1. What are some useful hints to help you debug a function or program?

Test small sections of a program in isolation.

Create repeatable errors based on different inputs, if they are reproducible, find the logic error.

Simple tests can be used to show things are handled incorrectly.

What happens in and around the point of error.

Use breakpoints and print statemetns.

Don’t assume that just because on input is handled correctly, your program is correct.

Use assertions and invariants to test

Think outside the box, never make any assumptions.

1. What is an assertion?

It is a usage of flow control or an assertion statement to terminate or control a program if a bound or other parameter is violated.

4. What is an invariant?

An invariant is similar to an insertion; however, it occurs within a loop.

1. Once you have identified assertions and invariants, how do you create a proof of correctness?

A proof of correctness is created via informal argument that explains why we think the code is correct and build the code around assertions. Once we use these assertions, we simulate the algorithm’s execution and if the assertions are valid, then the algorithm is proved as correct.

1. How is an assertion different from an assertion statement?

An assertion itself is simply a comment, usually used with a flow control bock, to state what will happen when a certain condition is met within the program. An assertion statement is an actual programming command used to terminate a program when the condition is not met or violated.

1. What is testing?

The process of going through a program and testing it for correctness. We do it to find errors and logical mistakes in the code, and fix them. Thorough testing is needed to create a robust program.

1. What is unit testing?

Unit testing is going through and testing each individual function for completeness and correctness.

1. What is a test harness?

A special main function/method, which is used purely for testing, is called a test harness. It will feed values into the function under test and then check the result.

1. What is boundary testing?

This is a process that tests the limits of the data, and will find out what the extent of the data functions.

1. What are some example boundary conditions?

The min of a function, the max of a function.

1. What is a test suite?

A collection of test cases, often involving a wide variety of simple tests for a function and program.

1. What is integration testing?

Integration Testing is used to check to make sure all the components of a program work together effectively.

1. What is regression testing and why is it performed?

If, during integration, issues are found in connected functions, we want to isolate them and remove them so the entire program can function properly.

1. What is black box testing?

Some testing considers only the structure of the input and output values, and ignores the algorithm used to produce the result

1. What is white box testing?

Consider the structure of the function, for example to ensure that every if statement is exercised both with a value that makes it true and a value that makes it false.

1. Give an informal description in English (not code) explaining how the bubble sort algorithm operates.

The bubble sort algorithm compares it’s predecessor value with itself, and then swaps them if necessary. It will continue to do this until the array is in order – however, this will take an excessive amount of time for a larger array and is just not recommended.

1. Give a similar description of the selection sorting algorithm.

This will find the largest element in an array, and swap that value into the final index. Then the next largest, and the next, until you reach the smallest element in the array.

1. In what ways does the analysis of recursive algorithms mirror the idea of mathematical induction?

It requires us to look at the base case and each step as we go through the array. Each step reduces other inputs until they reach a base case.

Chapter IV – Big-O(h) Notation

Study Questions 1.

1. **What is a linear search and how is it different from a binary search?**

Linear search is a form of 1 to 1 search, where it evaluates each of the items in an array or database. A binary search breaks down the search area into even units and looks for the desired value by continuously reducing the area searched for.

Linear searches compare everything, while binary searches break things down into smaller parts to find a thing more quickly.

1. **Can a linear search be performed on an unordered list? Can a binary search?**

Both can be used, but a linear search can take an excessive amount of time to complete, especially on a large list. A binary search can also be used by dividing things, because binary is n/2, where as linear is n. The amount of time to do the linear search increases proportionally to the number of items in a list, whereas a binary reduces it by half in proportion.

1. **If you start out with n items and repeatedly divide the collection in half, how many steps will you need before you have just a single element?**

It will remain proportional to the size of the set. So, if you have 1000 elements, it will take about 11 steps. (1000, 500, 250, 125, 63, 32, 16, 8, 4, 2, 1)

1. **Suppose an algorithm is O(n), where n is the input size. If the size of the input is doubled, how will the execution time change?**

It will double in proportion to the size of the input, as O(n) implies a linear algorithm.

1. **Suppose an algorithm is O(log n), where n is the input size. If the size of the input is doubled, how will the execution time change?**

O(log n) – it is proportional to the change, but will not double, instead it will follow a curve that maintains proportionality to the power of the change, so for example, if you have 8 items and it takes 3 seconds to go through, 16 items will take 4, 32 will take 5 and so on. It is relative to its exponent.

1. **Suppose an algorithm is O(n2 ), where n is the input size. If the size of the input is doubled, how will the execution time change?**

This is a quadratic algorithm, and thus will increase exponentially by a factor of 2 for every time it is doubled.

1. **What does it mean to say that one function dominates another when discussing algorithmic execution times?**

The largest (most time consuming) of the function/algorithm parts will dominate as the next size down is trivial compared to the next size up. For example, if a function takes 2.8 hours (n^2) and 1.2 seconds O(n log n), the amount of time in the equation O(n^2 + n log n) is trivial in the lesser part and thus not recognized for most intents and purposes. The highest time execution is always the one considered for a function and thus dominates.

1. **Explain in your own words why any sorting algorithm that only exchanges values with a neighbor must be in the worst case O(n2 ).**

Due to the nature of step by step swapping, it is extremely time consuming and MUST go through the data set each time to verify correctness of order. As a result, what may otherwise take a few seconds under O(n log n) will take hours, if not days, under a worst case O(n2). It results in 1+2+3+….n-1 + n behavior.

1. **Explain in your own words how the shell sort algorithm gets around this limitation.**

To work around the limit, shell sorting jumps locations within a search, to find the proper order for an array. It uses an extra loop to work as a ‘gap’ loop in the insertion sort. And thus, when it is sorting ten elements, it will jump, maintaining the proper order for numbers that are already in their proper spot and jump to the nearest out of place number. This makes things run much faster than a standard bubble or insertion sort.

1. **Give an informal description, in English, of how the merge sort algorithm works.**

A merge sort does divide and conquer, which looks through the sorted arrays, looks through them and then places everything into their proper order and merges them. It will continuously split the arrays it is given in half, until the order is created and implemented.

1. **What is the biggest advantage of merge sort over selection sort or insertion sort? What is the biggest disadvantage of merge sort?**

Insertion and selection sorts are O(n^2) worst case, thus can be very slow and take an excessive amount of time to perform their function. As a merge sort uses a divide and conquer technique, it can cut the amount of time to do a sort down drastically. However, it cannot do unsorted arrays very well and must evaluate everything at each step.

1. **In your own words given an informal explanation of the process of forming a partition.**

Partitioning occurs when you divide sections of array into two disparate parts. The limits are determined by the low and high (usually seen as min or max), and the third element known as the pivot. The pivot is the point of swap, where this location is exchanged with the first position. Invariants are preserved and all elements with index values that are smaller than the loop variable are smaller and equal to the pivot, whereas those of the inner loop will be larger. If it is smaller, then preserve the invariant and advance the index element. (i)

1. **Using the process of forming a partition described in the previous question, give an informal description of the quick sort algorithm.**

The quick sort goes through the array(s) and recursively goes through various partitions created by the algorithm and performs divide and conquer. There is a stark difference ot the merge sort. Whereas the merge sort devotes little effort to the partitioning of the array, the quick sort spends most of its time on that portion, by creating a single element pivot. It then divides the elements, into larger and smaller and equal portions and organizes the arrays that way.

1. **Why is the pivot swapped to the start of the array? Why not just leave it where it is? Give an example where this would lead to trouble.**

The pivot is swapped to the start of the array, and it moves the value out of any proportioning or break up steps. It helps to set the invariant, and preserves the pivot and its extended values. It only works for non-null comparable, and might run into issues where you have multiple equal values.

1. **In what ways is quick sort similar to merge sort? In what ways are they different?**

Both use divide and conquer. The merge sort; however, devotes little effort to the partitioning of the problem, whereas the quick sort devotes nearly all of its effort to this segment.

1. **What does the quick sort algorithm do if all elements in an array are equal? What is the big-Oh execution time in this case?**

Much like a null-case, this is a bad thing. The Big O execution time is O(n^2). If all are equal, this is also the case. In a merge sort, we would just have O(n log n) which is not terrible, but less than ideal.

1. **Suppose you selected the first element in the section being sorted as the pivot. What advantage would this have? What input would make this a very bad idea? What would be the big-Oh complexity of the quick sort algorithm in this case?**

It puts the smaller element in the front and makes it so there is less need for a separate array.

Bad Idea: Null Case, that would be O(n^2).

1. **Compare the partition median finding algorithm to binary search. In what ways are they similar? In what ways are they different?**

They divide things in two. That’s all I can think of at the moment. The different is rather than going by splits, the median finding algorithm might go for divide and conquer.