**Course code: Course title**
INFO6205: Program Structure and Algorithms

**Course Description**
This course covers the fundamentals of designing data structures and the algorithms which manipulate them. This is an important class for any aspiring developer as data structures and algorithms are at the core of every application. The goal of the course is not only to teach the fundamentals of the subject, but also to give an understanding of why.

**Course Outcomes**
A. Develop an approach to problem-solving.
B. Utilize different strategies to develop and improve various algorithms.
C. Articulate the relationship between data structures, algorithms, and invariants.
D. Choose the most effective utility (e.g., quicksort) for a particular programming task.
E. Prepare for a coding interview.

**Required Materials**
- Java Integrated Development Environment: here are installation instructions

**Welcome Module**

**Welcome to INFO 6205**

**Program Structure and Algorithms**

**What is this class all about?**
- Mostly it’s about solving problems by computer.
- You will learn about data structures, the rules that govern them (invariants), and the algorithms that operate on them.
- We will take a practical approach, that’s to say we will treat the subject matter with an “engineering” mindset.
- We will get a little mathematical where it’s appropriate because the foundation of everything we cover is, essentially, mathematics.

**Search Problems**
- Most of the problems we set out to solve will be “search problems.”
- In computer science terminology, a search problem is one where we will look for a solution within a potentially vast search space until we find one which satisfies our criteria.
- An example would be sorting: the solution we seek is one where all of the elements are in some predefined order.
- Of course, this is just an idealized way of classifying these problems—we are usually a lot smarter than that.
Practicalities

- The implementation language of this online version of the class is Java (8 or later). The subject material should ideally be independent of any particular language. But for practical reasons, we use Java. Examples and assignments are found in the github repository: https://github.com/rchillyard/INFO6205 (Java classes only).

Module 1: Solving Problems: The Use of Data Structures, Algorithms, and Invariants

Module Overview

In this module, we will explore the nature of the kinds of problems we can solve using computers. We will learn about different strategies for solving these problems: identifying invariants, using brute force, dividing or simplifying the problem into easier problems, and taking advantage of any existing order.

Learning Objectives

By the end of this module, you will be able to do the following:

- Develop techniques for solving problems.
- Detect invariants when solving problems.
- Apply the concept of reduction at an elementary level.
- Prove a recurrence relation by mathematical induction.
- Sort a small array using insertion sort.
- Identify a “search problem,” and show how a dictionary saves time.

Reading and Resources

<table>
<thead>
<tr>
<th>Required Resources</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntelliJ IDEA</td>
<td>Install the latest community edition.</td>
</tr>
<tr>
<td><a href="https://github.com/rchillyard/INFO6205">https://github.com/rchillyard/INFO6205</a></td>
<td>Clone the repository and run edu.neu.coe.info6205.sort.simple.InsertionSortTest. Study the insertion sort method being tested.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optional Resources</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://youtu.be/ROalU379l3U">https://youtu.be/ROalU379l3U</a></td>
<td>Insert-sort with Romanian folk dance</td>
</tr>
<tr>
<td><a href="https://youtu.be/QvuQH4_05LI">https://youtu.be/QvuQH4_05LI</a></td>
<td>ThreeBlueOneBrown on Problem-solving</td>
</tr>
</tbody>
</table>

Module Content and Tasks

**Title:** 1.1 Solving Puzzles

**Lesson Description:** In this lesson, we will take a look at strategies for solving problems and puzzles by computer - with special emphasis on the concept of invariants.
**Title:** 1.2 Recursion and Reduction  
**Lesson Description:** In this lesson we take an early look at the concept of reduction, which is fundamental to the efficient solution of most computer problems. And, we examine one of the most common strategies for implementing such solutions: recursion.

**Title:** 1.3 Search Problems  
**Lesson Description:** We learn some definitions in this lesson, including that of a search problem.

**Title:** 1.4 Sorting  
**Type:** Embedded Slideshow or link to PowerPoint file  
**Lesson Description:** This lesson introduces sorting using insertion sort as an exemplar.

**Assignment**

<table>
<thead>
<tr>
<th>Title</th>
<th>Prompt</th>
</tr>
</thead>
</table>
| **Module 1 Coding Assignment** | This assignment will start building your algorithmic skills, albeit we haven't really got into any serious algorithms yet. It will also give you some experience with random number generation in Java.  

An important type of practical experiment is normally referred to as the "random walk" experiment.  

Imagine a drunken man who, starting out leaning against a lamp post in the middle of an open space, takes a series of steps of the same length \( l \). The direction of these steps is randomly chosen from North, South, East or West. **After \( n \) steps, how far (\( d \)), generally speaking, is the man from the lamp post?** Note that \( d \) is the Euclidean distance of the man from the lamp-post.  

It turns out that there is a relationship between \( d \), \( l \) and \( n \) which is typically applicable to many different types of stochastic (randomized) experiments. Your task is to implement the code for the experiment and, most importantly, to **deduce the relationship**.  

Please clone/pull from the class repository and work on `RandomWalk.java` and `RandomWalkTest.java` each of package `randomwalk` and each under the appropriate source directory. (You may have to remove other java files from the classpath in order to allow the whole project to compile, but hopefully not.) Once you have all the unit tests running, you can do the experiment by running `RandomWalk` as a main program: provide the value of \( n \) as the first argument.  

For this particular assignment, it is **necessary but not sufficient** to ensure that the unit tests in `RandomWalkTest` all run. You must demonstrate via image files, graphs, whatever, what experiments you made in order to come up with the required expression. You will run the experiment for at least six values of \( n \) and will run each of...
these at least five times. That's to say, you will run the program at least 30 separate times.

Feel free to change the main program so that it will run all your experiments in one shot instead of 30 different runs.

Your submission should include:

- Your **conclusion** about the relationship between \( d \), \( n \) and \( l \);
- Your **evidence** to support that relationship (screenshot and/or graph and/or spreadsheet);
- Your **code** (*RandomWalk.java* plus anything else that you changed or created);
- A **screen shot** of the unit tests all passing.

Please note: for this assignment, you do **not** need to set up github and push your files, as described in the general instructions for submission (*Submitting Assignments*). Note also that common sense should tell you how \( d \) varies with \( l \). Don’t spend a lot of time agonizing over this aspect of the assignment. What we are primarily interested in is how \( d \) varies with \( n \).

**Rubric:**
https://docs.google.com/document/d/1VOhGQijPcP8xQM9Sip9WPJ3QjLulsz9dL9d5sO-ygQ/edit?usp=sharing

---

**Module 2: Complexity**

**Module Overview**

In order to understand the behavior and thus the cost of running algorithms, it is important to be able to place bounds on the efficacy of a solution or the difficulty of a problem. These are the roles of the Big O and Big Omega functions respectively. So, we will develop polynomial functions of \( N \), the size of a problem.

Because these functions usually involve logarithms, we will start with an in-depth look at logarithms. Next, we must recognize that the time to solve a problem may have as much to do with its memory requirements as the number of instructions executed. But, we must recognize that predicting the behavior from a theoretical viewpoint is likely to be at best an approximation. To really understand, we must benchmark our algorithms, and prior to that, test them. Finally, coming full-circle to the minimum complexity of the problem itself, we need some way to estimate that complexity—and for that we will learn about information entropy.

**Learning Objectives**

By the end of this module, you will be able to do the following:

- Plan how to plant 10 trees & design a 10-stage rocket.
- Use an understanding of memory and processor cycles to improve running time.
- Time and test the performance of an algorithm.
- Classify problems according to their “order of growth.”
- Estimate the entropy of a search problem.

### Reading and Resources

<table>
<thead>
<tr>
<th>Required Resources</th>
<th>Description</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Optional Resources</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://youtu.be/v4cd1O4zkGw">https://youtu.be/v4cd1O4zkGw</a></td>
<td>Hacker Rank on Big-O notation (some cute graphics) 8:30</td>
</tr>
</tbody>
</table>

### Module Content and Tasks

**Title:** 2.1 Logarithms  
**Lesson Description:** A detailed look at the logarithm function—the importance of which cannot be overemphasized.

**Title:** 2.2 Space vs Time  
**Lesson Description:** Where does all the time go? This lesson looks at both our natural enemies: large numbers of instruction cycles and large numbers of memory accesses. Which is worse? The answer may surprise you.

**Title:** 2.3 Benchmarking and Test-Driven Development  
**Lesson Description:** For any serious study of the behavior of an algorithm, it’s important to understand as much as possible about the mathematical basis of the solution. However, it’s even more important to back this up with benchmarking—and of course testing. As we’ll see in this lesson, no change should ever be made to optimize software without both testing and benchmarking the result.

**Title:** 2.4 Asymptotic Notation  
**Lesson Description:** In this lesson, we will look at—and try to understand—the important concepts of Big-O, Big-Omega, and their relatives.

**Title:** 2.5 Entropy  
**Lesson Description:** No study of problem solving and its mathematical analysis can be complete without an understanding of the root cause of the complexity: entropy. Understanding entropy can help us determine the theoretically best possible solution.
<table>
<thead>
<tr>
<th>Assignment</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Module 2 Coding Assignment</strong></td>
<td>Your task for this assignment is in three parts.</td>
</tr>
</tbody>
</table>

- (Part 1) You are to implement the `repeat`, `getClock`, and `toMillisecs` methods of the `Timer` class. Please see the skeleton class in the repository. For context, the function to be timed, hereinafter the "target" function, is the `Consumer function fu` passed in to one or the other of the constructors of `Benchmark_Timer`. For example, you might create a function which sorts an array with `n` elements.

  The generic type `T` is that of the input to the target function.  
The first parameter to the `run` method is the parameter that will, in turn, be passed to target function. In the `runFromSupplier` method, `supplier` will be invoked each time to get a `t` which is passed to the `repeat` method of `Timer`.  
The second parameter to the `run/runFromSupplier` function (`m`) is the number of times the target function will be called.  
The return value from `run/runFromSupplier` is the average number of milliseconds taken for each run of the target function.  
Don't forget to check your implementation by running the unit tests in `BenchmarkTest`.

- (Part 2) Implement `sort` (in the `InsertionSort` class) by (a) getting the code from the slides; (b) looking it up on the internet; (c) simply looking up the insertion code used by `Arrays.mergeSort` when the number of elements to sort is small. You should use the `helper.swap` method. Run the insertion sort test of `Benchmarks`.

- (Part 3) Measure the running times of this sort, using four different initial array ordering situations: random, ordered, partially-ordered and reverse-ordered. I suggest that your arrays to be sorted are of type `Integer`. Use the doubling method for choosing `n` and test for at least five values of `n`. Draw any conclusions from your observations regarding the order of growth.

As usual, the submission will be your entire project (clean, i.e. without the target and project folders). There are stubs and unit tests in the repository.

Report on your observations and show screenshots of the runs and also the unit tests. Please note that you may have to adjust the
required execution time for the insertion sort unit test(s) because your computer may not run at the same speed as mine.

Further notes: you should use the `System.nanoTime` method to get the clock time. This isn't guaranteed to be accurate which is one of the reasons you should run the experiment several times for each value of \( n \). Also, for each invocation of `run`, run the given target function ten times to get the system "warmed up" before you start the timing properly.

The `Sort` interface takes care of copying the array when the `sort(array)` signature is called. It returns a new array as a result. The original array is unchanged. Therefore, you do not need to worry about the insertion-based sorts getting quicker because of the arrays getting more sorted.

Submit by X day at 11:59 p.m. ET.

Rubric: [https://docs.google.com/document/d/1xxB6oTvUwgL6TniO3atf0a33fo8I5uuTV-SsDIQZiE/edit?usp=sharing](https://docs.google.com/document/d/1xxB6oTvUwgL6TniO3atf0a33fo8I5uuTV-SsDIQZiE/edit?usp=sharing)
Module Content and Tasks

**Title:** 3.1 Abstract Data Types  
**Lesson Description:** We will learn about the importance of the abstract data type as a component of applications.

**Title:** 3.2 LinkedList/Stack  
**Lesson Description:** The first ADT we study is perhaps the most familiar: the linked list. We will learn how the stack is essentially the same thing but which presents a different application programming interface.

**Title:** 3.3 Bag  
**Lesson Description:** This lesson introduces the lesser-known but extremely important ADT called the Bag.

**Title:** 3.4 Queue  
**Lesson Description:** The double-ended queue (or deque) is also a very important ADT. Like its counterpart in normal life, it plays an essential role in matching supply with demand and ensuring that clients are served on a first-come first-served basis.

**Title:** 3.5 Priority Queue  
**Lesson Description:** A variation on the double-ended queue, the priority queue has surprisingly many applications, especially in the world of Big Data. Instead of serving on a first-come first-served basis, it serves the most important (highest priority) first.

**Title:** 3.6 Objects vs Primitives in Java  
**Type:** Embedded Slideshow or link to PowerPoint file  
**Lesson Description:** All languages have their idiosyncrasies and Java is no exception. But, if you strive to write very efficient code, you must understand distinctive aspects of the language covered in this lesson.

Assignment

<table>
<thead>
<tr>
<th>Title</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Module 3 Coding Assignment</strong></td>
<td>Your task for this module is</td>
</tr>
<tr>
<td></td>
<td>Step 1:</td>
</tr>
<tr>
<td></td>
<td>(a) Implement height-weighted Quick Union with Path Compression. For this, you will flesh out the class UF_HWQUPC. All you have to do is to fill in the sections marked with // TO BE IMPLEMENTED ... // ...END IMPLEMENTATION.</td>
</tr>
</tbody>
</table>
(b) Check that the unit tests for this class all work. You must show "green" test results in your submission (screenshot is OK).

Step 2:
Using your implementation of UF_HWQUPC, develop a UF ("union-find") client that takes an integer value n from the command line to determine the number of "sites." Then generates random pairs of integers between 0 and n-1, calling connected() to determine if they are connected and union() if not. Loop until all sites are connected then print the number of connections generated. Package your program as a static method count() that takes n as the argument and returns the number of connections; and a main() that takes n from the command line, calls count() and prints the returned value. If you prefer, you can create a main program that doesn't require any input and runs the experiment for a fixed set of n values. Show evidence of your run(s).

Step 3:
Determine the relationship between the number of pairs (m) generated to accomplish this (i.e. to reduce the number of components from n to 1) is \( \sim \frac{1}{2} n \ln n \) where \( \ln n \) is the natural logarithm of n? Justify your conclusion.

Don't forget to follow the submission guidelines. And to use sufficient (and sufficiently large) different values of n.

Rubric:
https://docs.google.com/document/d/1rbM3ONbQDyc1QBBClndKUYoHbX5GMwa_uQK0mCjrioE/edit?usp=sharing

Module 4: Sorting, Shuffling & Selection

Module Overview
The "Dictionary principle" allows us to speed up searching, providing that we have an efficient method of sorting. Sorting comprises a large proportion of this course because it is so important. We cover all of the most important sorting algorithms here, leaving a few of the more advanced ones to Module 8. Additionally, we look at shuffling, or un-sorting, and selection, which essentially is finding the kth element of a collection, without having to sort it first.

Learning Objectives
By the end of this module, you will be able to do the following:

- Code a typical comparison method for a class of objects and describe the ways by which objects may be compared.
- Enumerate the rules of total order.
- Code selection sort and describe its complexity.
- Code shell sort and describe its complexity.
- Code merge sort and describe its complexity.
- Code quicksort and describe its complexity.
- Code heap sort and describe its complexity.
- Describe the complexity of QuickSelect.
- Code the Knuth shuffle and describe its complexity.

### Reading and Resources

<table>
<thead>
<tr>
<th>Required Resources</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sedgewick/Wayne, <em>Algorithms 4/e</em>, 2 Sorting, 2.1 Elementary Sort</td>
<td>This section provide details description and analysis of elementary sort.</td>
</tr>
<tr>
<td>Sedgewick/Wayne, <em>Algorithms 4/e</em>, 2 Sorting, 2.2 Merge sort</td>
<td>This section focuses on Merge sort and various approaches of merge sort.</td>
</tr>
<tr>
<td>Sedgewick/Wayne, <em>Algorithms 4/e</em>, 2 Sorting, 2.3 Quicksort</td>
<td>This section explains multiple partitioning techniques and quicksort.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optional Resources</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://www.toptal.com/developers/sorting-algorithms">https://www.toptal.com/developers/sorting-algorithms</a></td>
<td>This page shows visualization of multiple sorting algorithms.</td>
</tr>
<tr>
<td><a href="http://me.dt.in.th/page/Quicksort/#disqus_thread">http://me.dt.in.th/page/Quicksort/#disqus_thread</a></td>
<td>In-depth discussion of quicksort.</td>
</tr>
<tr>
<td><a href="https://youtu.be/CmPA7zE8mx0">https://youtu.be/CmPA7zE8mx0</a></td>
<td>Shell Sort by folk dance using gaps of 5, 3, 1.</td>
</tr>
</tbody>
</table>

### Module Content and Tasks

**Title: 4.1 Comparison Sorts**

**Lesson Description:** General introduction to comparison sorts, invariants, and swaps.

**Title: 4.2 Selection Sort**

**Lesson Description:** Description of selection sort

**Title: 4.3 Shell Sort**

**Lesson Description:** Description of shell sort: how to reduce the number of inversions before sorting

**Title: 4.4 Merge Sort**

**Lesson Description:** Divide and conquer-based sorting: Merge sort
Title: 4.5 Quicksort  
Lesson Description: Divide and conquer-based sorting: Quicksort

Title: 4.6 Heap Sort  
Lesson Description: Binary heaps and heap sort.

Title: 4.7 Quickselect  
Lesson Description: Order statistics without getting a complete order.

Title: 4.8 Shuffling  
Lesson Description: Shuffling – the opposite of sorting.

Module Summary Chart  
Title: Module 4 Interactive Flowchart for Choosing a Sort Algorithm  
URL: https://drive.google.com/file/d/1e5DGj9xhprriiCNqAxAxPs9QWQmKAIQ/view?usp=sharing

Assignment

<table>
<thead>
<tr>
<th>Title</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module 4 Coding Assignment</td>
<td>Parallel Sort</td>
</tr>
</tbody>
</table>

Your task is to implement a parallel sorting algorithm such that each partition of the array is sorted in parallel. You will consider two different schemes for deciding whether to sort in parallel.

A cutoff (defaults to, say, 1000) which you will update according to the first argument in the command line when running. It's your job to experiment and come up with a good value for this cutoff. If there are fewer elements to sort than the cutoff, then you should use the system sort instead.

Recursion depth or the number of available threads. Using this determination, you might decide on an ideal number (t) of separate threads (stick to powers of 2) and arrange for that number of partitions to be parallelized (by preventing recursion after the depth of lg t is reached).

An appropriate combination of these.

There is a Main class and the ParSort class in the sort.par package of the INFO6205 repository. The Main class can be
used as is but the ParSort class needs to be implemented where you see "TODO..."

Unless you have a good reason not to, you should just go along with the Java8-style future implementations provided for you in the class repository.

You must prepare a report that shows the results of your experiments and draws a conclusion (or more) about the efficacy of this method of parallelizing sort. Your experiments should involve sorting arrays of sufficient size for the parallel sort to make a difference. You should run with many different array sizes (they must be sufficiently large to make parallel sorting worthwhile, obviously) and different cutoff schemes.

For varying the number of threads available, you might want to consult the following resources:


Rubric:
https://docs.google.com/document/d/12zvnyMaYl6D0E0GH0DVTCsT-YAKLdSagwBDC24o8qAI/edit?usp=sharing

Module 5: Searching

Module Overview
Information retrieval is one of the most important tasks for computers. How can we store information in such a way that retrieval of the information is fast and uses as little memory as is reasonably possible? How do we deal with objects that we can compare and objects that we cannot compare? How do we retrieve information when getting a block of data is extremely slow (as with a hard disk, or a network lookup, for instance)? These questions are addressed in this important module.

Learning Objectives
By the end of this module, you will be able to do the following:
- Code binary search and describe its complexity.
- Analyze the performance of a binary search tree.
- Differentiate the 2-3 tree from the left-leaning red-black tree.
- Identify situations for which a B-tree search is applicable, and state the advantages of that search.
- Discuss the required features of a hash function for searching.
- Code and analyze separate chaining.
- Code and analyze open addressing.
Reading and Resources

<table>
<thead>
<tr>
<th>Required Resources</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sedgewick/Wayne, <em>Algorithms 4/e</em>, sections 3.2</td>
<td>Detailed description of Binary search tree and its applications</td>
</tr>
<tr>
<td>Sedgewick/Wayne, <em>Algorithms 4/e</em>, sections 3.3, 3.4</td>
<td>Balanced Search Tree, Hash Table</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optional Resources</th>
<th>Description</th>
</tr>
</thead>
</table>

Module Content and Tasks

**Title: 5.1 Introduction to Searching**
**Lesson Description:** Searching basics: lists, arrays.

**Title: 5.2 Binary Search Trees**
**Lesson Description:** How an extra degree of freedom can allow us to efficiently store and retrieve keys.

**Title: 5.3 Balanced Search Trees**
**Lesson Description:** Adding another degree of freedom to allow us to balance a search tree.

**Title: 5.4 B-Trees**
**Lesson Description:** Extending the concept of balanced trees.

**Title: 5.5 Hash Functions**
**Lesson Description:** How to get the benefits of an array with generalized keys.

**Title: 5.6 Separate Chaining**
**Lesson Description:** Chaining: the simplest implementation of a hash table.

**Title: 5.7 Open Addressing**
**Lesson Description:** Also known as linear probing, a slightly more efficient hash table if it’s well managed.

Module Summary Chart
**Assignment**

<table>
<thead>
<tr>
<th>Title</th>
<th>Prompt</th>
</tr>
</thead>
</table>
| *Module 5 Coding Assignment* | **Hibbard deletion:**  
You will perform more experiments to test that the depth/height of a Binary Search Tree after M (Hibbard) deletions and insertions will be proportional to the square root of N where N is the size of the tree when M is large. The consequence of this is that deletion, search and insertion will all end up being $O(N^{1/2})$ instead of $O(\lg N)$ which is what we would prefer.

The coding you need to do for this project is to implement the various delete cases. Insert operations are already coded. Please see the package *edu.neu.coe.info6205.symbolTable*.

As usual, push your solution to your git repository for submission. Your submission should also include images of your test runs and any graphs or other evidence that you feel is appropriate.

(Note: You can refer to Sedgewick/Wayne, *Algorithms* 4/e, Section 3.2 Binary Search Trees for better understanding of deletion.)

Don't forget to follow the submission guidelines.

**Rubric:**
[https://docs.google.com/document/d/1NNmdQW0q1cOV8RymOlQuy0D0BaBP16HCTaGGsF6m70w/edit?usp=sharing](https://docs.google.com/document/d/1NNmdQW0q1cOV8RymOlQuy0D0BaBP16HCTaGGsF6m70w/edit?usp=sharing)

---

**Module 6: Compression**

**Module Overview**

Space (memory) can be as important as the number of instructions when it comes to the performance of algorithms. Compressing data for transmission over a network, for instance, can make a significant difference in the time taken for messages to be passed. We will look in detail only at Huffman coding, but the textbook also covers run-length encoding.

**Learning Objectives**

By the end of this module, you will be able to do the following:

- Describe the role of compression algorithms in data storage, communication, and processing.
- Encode and decode by Huffman coding.
Reading and Resources

<table>
<thead>
<tr>
<th>Required Resources</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sedgewick/Wayne, <em>Algorithms</em> 4/e, 5.5 Data Compression, pages 826-839</td>
<td>These pages explore Huffman Coding and tries.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optional Resources</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sedgewick/Wayne, <em>Algorithms</em> 4/e, 5.5 Data Compression, pages 810-850</td>
<td>This section additionally explores run-length encoding and LZW compression.</td>
</tr>
</tbody>
</table>

Module Content and Tasks

**Title**: 6.1 Huffman Codes

**Lesson Description**: Message compression by Huffman coding.

Assignment

<table>
<thead>
<tr>
<th>Title</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module 6 Coding Assignment</td>
<td>Huffman Encoding</td>
</tr>
<tr>
<td></td>
<td>As a part of this assignment you need to implement the most famous message compression algorithm, a Huffman coding. Refer the HuffmanCoding.java class from the repository and implement the following methods to implement Huffman coding.</td>
</tr>
<tr>
<td></td>
<td>[task1] First implement the add(int x) method which will be used in the encoder function.</td>
</tr>
<tr>
<td></td>
<td>[task2] You need to implement the dfs(BiFunction&lt;Object, Integer, Object&gt; depthFunction, BiConsumer&lt;Node, Object&gt; consumer, Object depthIndicator, Integer branch) function which will be used in Encoder. This dfs function is used in creating the tree which will be used in encoding.</td>
</tr>
<tr>
<td></td>
<td>[task3] You need to implement the getEncoder(Node node) by using the above dfs function to create the map of the symbol(characters) and their frequency.</td>
</tr>
<tr>
<td></td>
<td>[task4] Finally you need to implement function decode(StringBuilder stringBuilder, Node state, long x) to decode the encoded message. Don’t forget to use the mask variable while implementing the decode function.</td>
</tr>
</tbody>
</table>

The assignment might appear little difficult but if you go through the reference material, you would find it little easy.

Rubric:
Module 7: Graphs and Trees

Module Overview
This module covers those aspects of graph theory suitable for a general course such as INFO6205. Graphs are increasingly important data structures for the modern world and, because graphs tend to be very large in terms of the numbers of edges, it is extremely important to design efficient algorithms.

Learning Objectives
By the end of this module, you will be able to do the following:
- Describe the importance and applications of graphs and trees.
- Differentiate between a tree and a graph which is not a tree.
- Code and analyze depth-first and breadth-first traversal of trees and graphs.
- Differentiate between undirected and directed graphs.
- Design data structures for undirected and directed graphs.
- Explain how the connected components algorithm is used to compute connected components for a given graph.
- Code and analyze Kosaraju’s algorithm.
- Code and analyze Dijkstra’s algorithm.

Reading and Resources

<table>
<thead>
<tr>
<th>Required Resources</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://www.maa.org/press/periodicals/convergence/leonard-eulers-solution-to-the-konigsberg-bridge-problem">https://www.maa.org/press/periodicals/convergence/leonard-eulers-solution-to-the-konigsberg-bridge-problem</a></td>
<td>The Seven Bridges of Königsberg</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optional Resources</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www1.coe.neu.edu/~rhillyard/">http://www1.coe.neu.edu/~rhillyard/</a> Tunnel%20article.pdf</td>
<td>Application of Kruskal’s algorithm--part of lesson 7.5</td>
</tr>
<tr>
<td><a href="https://tinkerpop.apache.org">https://tinkerpop.apache.org</a></td>
<td>Apache Tinker Pop Graph Computing Framework</td>
</tr>
</tbody>
</table>

Module Content and Tasks

**Title:** 7.1 Introduction to Trees and Graphs  
**Lesson Description:** General description of trees and graphs.

**Title:** 7.2 Depth-first vs Breadth-first  
**Lesson Description:** The two primary strategies for traversing graphs and trees.
**Title:** 7.3 Undirected and Directed Graphs  
**Lesson Description:** Graph properties and algorithms.

**Title:** 7.4 Connected and Strong Components  
**Lesson Description:** Connectivity in graphs – undirected and directed.

**Title:** 7.5 Minimum Spanning Trees  
**Lesson Description:** How to define a minimum spanning tree.

**Title:** Kruskal’s Algorithm, NEU Tunnel System  
**Type:** Explainer Video

**Title:** 7.6 Shortest Paths  
**Lesson Description:** Calculating the shortest path(s) from A to B using Dijkstra’s algorithm.

### Assignment

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Prompt</th>
</tr>
</thead>
</table>
| **Module 7 Coding Assignment** | **Assignment A:** Given a 2d Matrix which is a map of ‘1’s (rocks) and ‘0’s (water), count the number of rocky islands. A rocky island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.  

Task: Complete the function `islands(int[][] grid)` and return the count. You can create a separate function on the top of the given function to count the islands. This problem can have multiple solutions. Any one solution is acceptable.  

**Assignment B:**  
A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible.  

You have been given the task of designing tunnels to connect all of the buildings in a given list with the lowest cost, where cost is proportional to the total length. You need to implement the **TODOs (TO BE)**
Module 8: Advanced Sorts

Module Overview
This module covers advanced sorts--mostly sorts that either don’t use comparison at all or, if they do, use it in some hybrid algorithm. Module 8 also introduces the Group Project.

Learning Objectives
By the end of this module, you will be able to do the following:
- Code and analyze pre-sorting algorithms.
- Code and analyze non-comparison-based sorting.
- Demonstrate teamwork and cooperation skills.

Task List
- Lesson 8.1 Intro Video, lesson content, and Check Your Knowledge
- Lesson 8.2 Intro Video, lesson content, and Check Your Knowledge
- Coding Assignment
- Summary Video
- Start Team Project

Module Content and Tasks
Title: 8.1 Bucket Sort and Husky Sort
Lesson Description: Hybrid sorts

IMPLEMENTED) in these files: Bag_Array.java, UF_HWQUPC.java, Kruskal.java. Feel free to add helper functions if you need any. There is a Tunnels Class in the edu.neu.coe.info6250.graphs.tunnels package, you can use the main() method to get the lowest cost set of tunnels. As usual, the submission will be your entire project (clean, i.e. without the target and project folders). Report on your observations(all of the tunnels you've built, the total cost and total length) and show screenshots of the runs and also the unit tests.
Module 9: Finite Automata & Classifying Search Problems

Module Overview
This module covers some advanced theoretical topics that don’t fit neatly into any of the other modules.

Learning Objective
By the end of this module, you will be able to do the following:
- Code and use regular expressions.
- Understand the relationship between regular expressions and finite automata.
- Distinguish between P, NP, etc.
- Demonstrate teamwork and cooperation skills.

Reading and Resources

<table>
<thead>
<tr>
<th>Required Resources</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sedgewick/Wayne, <em>Algorithms</em> 4/e, 5.4</td>
<td>These pages explore Regular Expressions, as well as deterministic and non-deterministic finite automata.</td>
</tr>
<tr>
<td>Regular Expressions, pages 788-809</td>
<td></td>
</tr>
<tr>
<td>Sedgewick/Wayne, <em>Algorithms</em> 4/e,</td>
<td>Reduction, Intractability</td>
</tr>
<tr>
<td>Reduction, pages 903-921</td>
<td></td>
</tr>
</tbody>
</table>

Module Content and Tasks

Lesson 9.1
Title: Regular Expressions
Lesson Description: Finite State Machines and their application to parsing.

Lesson 9.2
Title: DFAs and NFAs
Lesson Description: Deterministic and non-deterministic finite automata

Lesson 9.3
Title: Classifying Search Problems
Lesson Description: P and NP

Project and Optional Module 10: Advanced Problem Solving

Module Overview
This module covers specialized topics that are not treated in standard textbooks but which you should have at least some knowledge about.

**Learning Objectives**

By the end of this module, you will be able to do the following:

- Design a non-deterministic algorithm for solving an NP-complete problem.
- Differentiate between greedy algorithms and dynamic programming.
- Demonstrate teamwork and cooperation skills.

**Reading and Resources**

<table>
<thead>
<tr>
<th>Required Resources</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sedgewick/Wayne, <em>Algorithms 4/e</em>, pages tk</td>
<td>These chapters explore ___.</td>
</tr>
</tbody>
</table>

**Module Content and Tasks**

**Lesson 10.1**

**Title**: Non-deterministic Programming  
**Lesson Description**: Introduction to Genetic Algorithms

**Lesson 10.2**

**Title**: Dynamic Programming  
**Lesson Description**: A quick look at dynamic programming

**Module 11: Guide to Interviews**

**Module Overview**

This module does not directly relate to anything in the textbook, nor will it appear in the final exam. Interviews for software development positions have become harder and more competitive in recent decades. Gone are the days when the interviewer would ask just one or two softball questions about languages you claimed to know. In short, you are expected to apply everything you learned in this course during an intense verbal examination that might last an hour. There is no substitute for practice. The more you solve internet problems, the more comfortable you will be under pressure. This module gives you a few pointers to help you when the moment comes.

**Learning Objectives**

By the end of this module, you will be able to do the following:

- Conduct an appropriate coding interview.
- Debug your own programs.
- Apply problem-solving skills to a coding case study.
- Demonstrate teamwork and cooperation skills.

**Reading and Resources**
<table>
<thead>
<tr>
<th>Required Resources</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Optional Resources</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://www.amazon.com/dp/0984782850/ref=cm_sw_em_r_m_t_dp_U_2fdhFbEWAE53S">https://www.amazon.com/dp/0984782850/ref=cm_sw_em_r_m_t_dp_U_2fdhFbEWAE53S</a></td>
<td>Cracking the Coding Interview by Gayle Laakman McDowell</td>
</tr>
<tr>
<td><a href="https://medium.com/@AndyyHope/software-engineering-interviews-744380f4f2af">https://medium.com/@AndyyHope/software-engineering-interviews-744380f4f2af</a></td>
<td>Excellent article on preparing for interviews at companies like Facebook.</td>
</tr>
<tr>
<td><a href="https://en.wikipedia.org/wiki/The_Hitchhiker's_Guide_to_the_Galaxy">https://en.wikipedia.org/wiki/The_Hitchhiker's_Guide_to_the_Galaxy</a></td>
<td>The increasingly inaccurately described Hitchhikers’ trilogy. The cover of the book is DON’T PANIC</td>
</tr>
</tbody>
</table>

**Module Content and Tasks**

**Lesson 11.1**

*Title*: DON’T PANIC  
*Lesson Description*: The Hitchhiker’s Guide to Interviews 11.1

**Lesson 11.2**

*Title*: Interview Case Studies  
*Lesson Description*: A look at one or two example problems

**Module 12: Final Exam & Submit Team Project**