Chapter 18 - **Recursion**
18.1 (720) - **Introduction** - Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops.
18.3 (723-726) - **Case Study: Computing Fibonacci Numbers** - In some cases, recursion enables you to create an intuitive, straightforward, simple solution to a problem.
Compute Fibonacci (724)
18.4 (726) - **Problem Solving Using Recursion** -All recursive methods have the following characteristics:
- The method is implemented using an if-else or a switch statement that leads to different cases.
- One or more base cases (the simplest case) are used to stop recursion.
- Every recursive call reduces the original problem, bringing it increasingly closer to a base until it becomes that case.

RecursivePalindromeUsingSubstring (727-728)
18.5 (728-730) - **Recursive Helper Methods** - Sometimes you can find a solution to the original problem by defining a recursive function to a problem similar to the original problem. This new method is called a recursive helper method. The original problem can be solved by invoking the recursive helper method.
- A second method that receives additional parameters is known as a recursive helper method.

Recursive Palindrome (728-729)
Recursive Selection Sort (729) - Find the smallest element in the list and swap it with the first element, ignore the first element and sort the remaining smaller list recursively.
Recursive Binary Search (730) - Case 1: The key is less than the middle element, recursive search for the key in the first half of the array.
Case 2: If the key is equal to the middle element, search ends with a match.
Case 3: If the key is greater than the middle element, recursive search for the key in the second half of the array.
18.9 (740) - **Recursion vs. Iteration** - Recursion is an alternative form of program control. It is essentially repetition without a loop.
- Recursion bears substantial overhead.
- Recursion uses too much time and too much memory
- Recursion enables you specify a clear, simple solution for an inherently recursive problem that would otherwise be difficult to obtain.
- Use whichever approach can best develop an intuitive solution that naturally mirrors the problem
- If an iterative solution is obvious, use it because it will generally be more efficient than the recursive option.
- Recursive programs can run out of memory, causing a StackOverflowError

18.10 (740-741) - **Tail Recursion** - A tail-recursive method is efficient. A recursive method is said to be tail recursive if there are no pending operations to be performed on return from a recursive call. Some compilers can optimize tail recursion to reduce stack size.

Chapter 19 - **Generics**
19.1 (752) - **Introduction** - Generics enable you to detect errors at compile rather than at runtime.
19.2 (752-754) - **Motivations and Benefits** - The motivation for using Java generics is to detect errors at compile time.
- A generic class of method permits you to specify allowable types of objects that the class or method can work with.
- formal generic type - <E> which can be replaced later with an actual concrete type
- generic instantiation - replaying a generic type
- ArrayList and ArrayList<E> methods (pg. 753)

19.3 (754-756) - **Defining Generic Classes and Interfaces** - A generic type can be defined for a class or interface. A concrete type must be specified wen using the class to create an object or using the class or interface to declare a reference variable.
GenericStack<E> (pg. 755)
19.4 (756-758) - Generic Methods - A generic type can be defined for a static method.
- public static <E> void print(E[] list)
- GenericMethodDemo.<Integer>print(integers);
- print(integers)

Bounded generic type: <E extends GeometricObject> specifies that E is a generic subtype of GeometricObject
- An unbounded generic type <E> is the same as <E extends Object>

To define a generic type for a class, place it after the class name, such as GenericStack<E>.
To define a generic type for a method, place the generic type before the method return type, such as <E> void max (E o1, E o2)
19.5 (758-759) - **Case Study: Sorting an Array of Objects** - You can develop a generic method for sorting an array of Comparable elements.
- public static <E extends Comparable<E>> void sort(E[] list) {}
- generic sort method (759)

Chapter 22 -**Developing Efficient Algorithms**
22.1 (840) - **Introduction -** Algorithm design is to develop a mathematical process for solving a problem. Algorithm analysis is to predict the performance of an algorithm.
22.2 (840-841) - **Measuring Algorithm Efficiency Using Big O Notation** - The Big O notation obtains a function for measuring algorithm time complexity based on the input size. You can ignore multiplicative constants and non dominating terms in the function.
- Compare two algorithms by their growth rates
- *Time complexity* is a measure of execution time using the Big O notation
- *Space complexity* measures the amount of memory space used by an algorithm.

22.3 (842-844) - **Examples Determining Big O**
- a loop is n and the action in the loop is a constant
- two separate loops, add
- adding O(n) + O(n) = O(n)

Mathematical summations used in algorithm analysis: (841)
22.4 (846-848) **Analyzing Algorithm Time Complexity**
22.4.1 Analyzing Binary Search (846)
22.4.2 Analyzing Selection Sort (846)
22.4.3 Analyzing the Tower of Hanoi Problem (846-847)

TABLE 22.2 Common Recurrence Functions

Recurrence Relation	*Result*	*Example*
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$	Binary search, Euclid's GCD
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$	Linear search
$T(n) = 2T(n/2) + O(1)$	$T(n) = O(n)$	CheckPoint Question 22.8.2
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$	Merge sort (Chapter 23)
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort
$T(n) = 2T(n-1) + O(1)$	$T(n) = O(2^n)$	Tower of Hanoi
$T(n) = T(n-1) + T(n-2) + O(1)$	$T(n) = O(2^n)$	Recursive Fibonacci algorithm

22.4.4 Common Recurrence Relations
22.4.5 Comparing Common Growth Functions
O(1) < O(logn) < O(n) < O(n logn) < O(n^2) < O(n^3) < O(2^n)

Chapter 20 - **Lists, Stacks, Queues, and Priority Queues**
20.1 (776) **Introduction** - Data structures
- data structure - collection of organized data, stores data and supports operations for accessing and manipulating the data, or *containers*
- Java Collections Framework - data structures, organization and manipulation

ArrayList and LinkedList
20.2 (776-780) - **Collections** - The Collection interface defines the common operations for lists, vectors, stacks, queues, priority queues, and sets. A collection is a type of container.
Collection interface methods (778)
If a method in the Collection interface cannot be implemented in the concrete subclass, then an UnsupportedOperationException is thrown
20.3 (780-781) **Iterators** - Each collection is Iterable. You can obtain its Iterator object to traverse all the elements in the collection.
Collection<String> collection = new ArrayList<>();
Iterator<String> iterator = collection.iterator()
20.4 (781-782) **Using the forEach Method** - use the forEach method to perform an action for each element in a collection.
collection.forEach(e -> System.out.print(e.toUpperCase() + " "));
20.5 (782-786) **Lists** - The List interface extends the Collection interface and defines a collection for storing elements in a sequential order. (asList() method pg. 786)
ArrayList and LinkedList
- methods for List interface (783)
- methods for ListIterator interface (783)

Linked List vs. ArrayList:
- LinkedList: insertion or deletion of elements at the beginning of the list
- ArrayList: support random access through an index without inserting or removing elements at the beginning of the list
- **Creating an ArrayList & methods (784)**
- **Creating a LinkedList & methods (785)**

20.6 (787-790) **The Comparator Interface** - Comparator can be used to compare the objects of a class that doesn't implement Comparable or define a new criteria for comparing objects.
GeometricObjectComparator (787) - Writing a comparator class that implements the comparator class, example of writing a comparison check
- Comparing elements using the Comparable interface is referred to as comparing using natural order.
- Comparing elements using the Comparator interface is referred to as comparing using comparator.

Define a custom comparator (789)
20.7 (791-794) **Static Methods for Lists and Collections** - The Collections class contains static methods to perform common operations in a collection and a list.
Static methods for list and collection (791)
20.9 (798-799) Vector and Stack Classes - Vector is a subclass of AbstractList and Stack is a subclass of Vector in the JavaAPI.
- vector methods (798), stack methods (799)

20.10 (799-802) **Queues and Priority Queues** - In a priority queue, the element with the highest priority is removed first.
- The Queue Interface methods (800)
- deque is a double ended queue, methods for a deque (801 top)

Chapter 24 - **Implementing Lists, Stacks, Queues, and Priority Queues**
24.1 (918) - **Introduction** - Implementing Data Structures
24.2 (918-922) - **Common Operations for Lists** - Common operations of lists are defined in the List interface.
- A list is a popular data structure for sorting data in sequential order
- Operations for a list:
  - Retrieve an element from the list
  - Insert a new element into the list
  - Delete an element from the list
  - Find out how many elements are in the list
  - Determine whether an element is in the list
  - Check whether a list is empty

MyList interface and methods (920 top)
24.3 (922-928) **Array Lists** - An array list is implemented using an array.
- **24.3.1 MyArrayList<E> creation and methods (923) (innards of the methods 923-926)**

24.4 (929-942) Linked Lists - A linked list is implemented using a linked structure.
- 24.4.1 Nodes - In a linked list, each element is contained in an object called *node*.
- **24.4.2 MyLinkedList<E> creation and methods (931)**
- 24.4.3 Implementing MyLinkedList
  - addFirst(e) method (933 top)
  - addLast(e) method (934 top)
  - add(index, e) method (934 bottom)
  - removeFirst() method (935 bottom)
  - removeLast() method (936 bottom)
  - remove(index) method (937 bottom)

Listing 24.5 MyLinkedList.java - setup and innards of implemented methods (938 bottom)
24.4.6 **MyArrayList vs. MyLinkedList**
- Both are used to store a list.
- MyArrayList is implemented using an array vs. MyLinkedList is implemented using a linked list
- Overhead: MyArrayList < MyLinkedList
- Efficient in terms of inserting elements into and delete elements from the beginning of the list: MyLinkedList

TABLE 24.1 Time Complexities for Methods in `MyArrayList` and `MyLinkedList`

Methods	*MyArrayList/ArrayList*	*MyLinkedList/LinkedList*
`add(e: E)`	$O(1)$	$O(1)$
`add(index: int, e: E)`	$O(n)$	$O(n)$
`clear()`	$O(1)$	$O(1)$
`contains(e: E)`	$O(n)$	$O(n)$
`get(index: int)`	$O(1)$	$O(n)$
`indexOf(e: E)`	$O(n)$	$O(n)$
`isEmpty()`	$O(1)$	$O(1)$
`lastIndexOf(e: E)`	$O(n)$	$O(n)$
`remove(e: E)`	$O(n)$	$O(n)$
`size()`	$O(1)$	$O(1)$
`remove(index: int)`	$O(n)$	$O(n)$
`set(index: int, e: E)`	$O(n)$	$O(n)$
`addFirst(e: E)`	$O(1)$	$O(1)$
`removeFirst()`	$O(n)$	$O(1)$

24.4.5 Variations of LinkedLists (942)
- Doubly, circular and circular doubly linked lists.

24.5 (943-946) **Stacks and Queues** - Stacks can be implemented using array lists and queues can be implemented using linked lists.
GenericQueue<E> methods and creation (945)
24.6 (947-948) Priority Queues - Priority queues can be implemented using heaps.
MyPriorityQueue<E extends Comparable <E>> methods and creation (947-948)