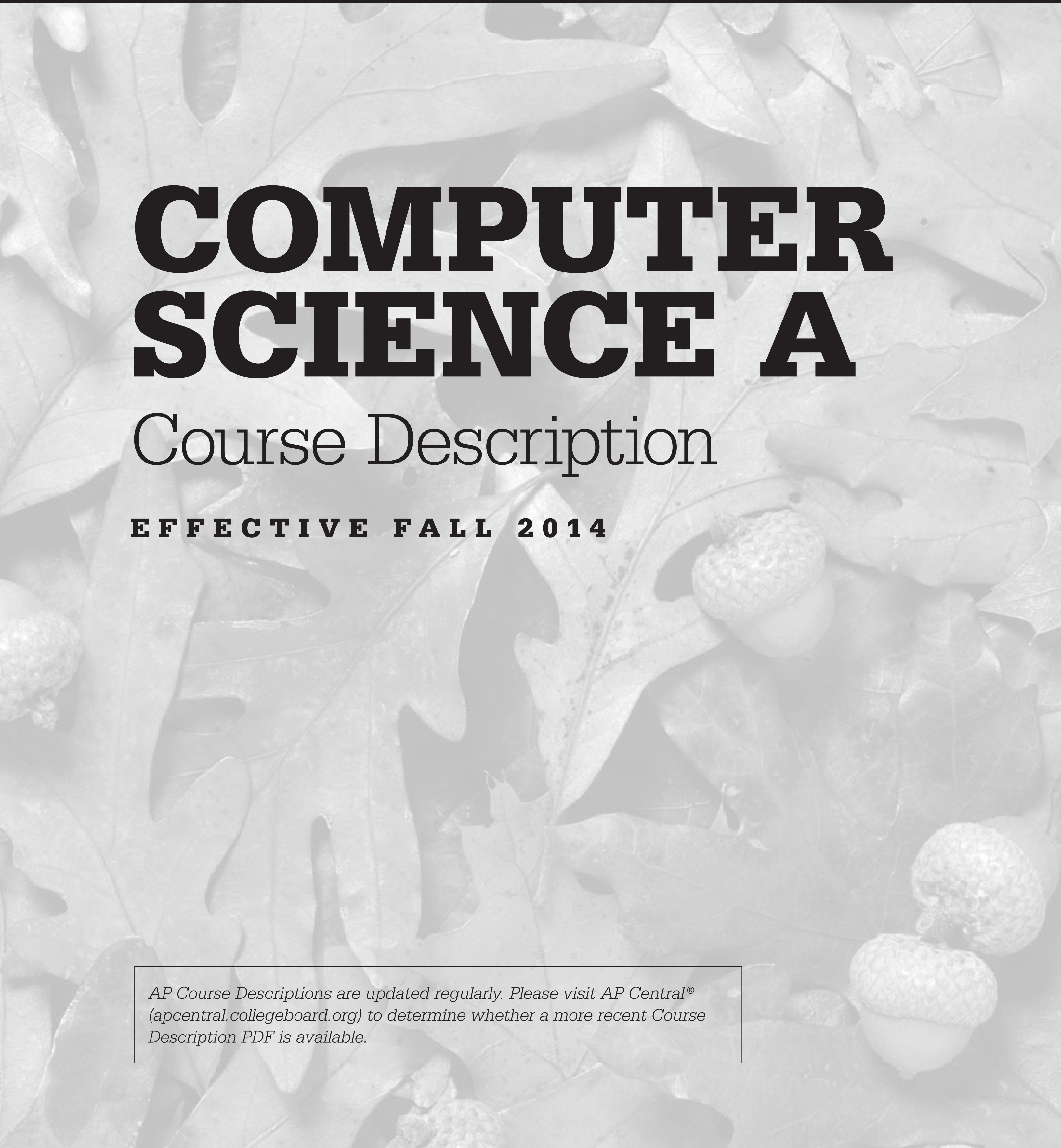
CollegeBoard AP



COMPUTER SCIENCE A

Course Description

EFFECTIVE FALL 2014

THE EXAM

The AP Computer Science A Exam is 3 hours long and seeks to determine how well students have mastered the concepts and techniques contained in the course outline. The exam consists of two sections: a multiple-choice section (40 questions in 1 hour and 30 minutes), which tests proficiency in a wide variety of topics, and a free-response section (4 questions in 1 hour and 30 minutes), which requires the student to demonstrate the ability to solve problems involving more extended reasoning.

The multiple-choice and the free-response sections of the AP Computer Science A Exam require students to demonstrate their ability to solve problems, including their ability to design, write, and analyze programs and subprograms. Minor points of syntax are not tested on the exam. All code given is consistent with the AP Java subset. All student responses involving code must be written in Java. Students are expected to be familiar with and able to use the standard Java classes and interfaces listed in the AP Java subset. For both the multiple-choice and the free-response sections of the exam, a quick reference to the classes and interfaces in the AP Java subset will be provided. The Java Quick Reference is included in Appendix B.

In the determination of the grade for the exam, the multiple-choice section and the free-response section are given equal weight. Because the exam is designed for full coverage of the subject matter, it is not expected that many students will be able to correctly answer all the questions in either the multiple-choice section or the free-response section in the time allotted.

Multiple-choice questions on the exam are classified according to the type of content that is tested in the question. Questions may be listed in one or more of the classification categories. For example, a question that uses a looping construct to traverse the elements of an array would be listed under both the Data Structures and the Programming Fundamentals categories. The table below shows the classification categories and how they are represented in the multiple-choice section of the exam. Because questions can be classified in more than one category, the total of the percentages is greater than 100%.

Classification Category	**Percent of multiple-choice items**
Programming Fundamentals	55–75%
Data Structures	20–40%
Logic	5–15%
Algorithms/Problem Solving	25–45%
Object-Oriented Programming	15–25%
Recursion	5–15%
Software Engineering	2–10%

APPENDIX A

AP Computer Science Java Subset

The AP Java subset is intended to outline the features of Java that may appear on the AP Computer Science A Exam. The AP Java subset is NOT intended as an overall prescription for computer science courses — the subset itself will need to be supplemented in order to address all topics in a typical introductory curriculum. For example, input and output must be part of a course on computer programming. However, there are many ways to handle input and output in Java. Because of this variation, details of input and output (except for basic text output using `System.out.print` and `System.out.println`) are not tested on the AP Computer Science A Exam.

This appendix describes the Java subset that students will be expected to understand when they take the AP Computer Science A Exam. A number of features are also mentioned that are potentially relevant in an introductory computer science course but are not tested on the exam. Omission of a feature from the AP Java subset does not imply any judgment that the feature is inferior or not worthwhile.

The AP Java subset was selected to

1. enable the test designers to formulate meaningful questions.
2. help students with test preparation.
3. enable instructors to follow a variety of approaches in their courses.

To help students with test preparation, the AP Java subset was intentionally kept small. Language constructs and library features were omitted that did not add significant functionality and that can, for the formulation of exam questions, be expressed by other mechanisms in the subset.

The AP Java subset gives instructors flexibility in how they use Java in their course. For example, some courses teach how to perform input/output using streams or readers/writers, others teach graphical user interface construction, and yet others rely on a tool or library that handles input/output. For the purpose of the AP Computer Science A Exam, these choices are incidental and are not central for the problem solving process or for the mastery of computer science concepts. The AP Java subset does not address handling of user input at all. That means that the subset is not complete. To create actual programs, instructors need to present additional mechanisms in their courses.

The following section contains the language features that may be tested on the AP Computer Science A Exam. The Java Quick Reference contains a list specifying which Standard Java classes, interfaces, constants, and methods may be used on the exam. This document is available to students when they take the exam, is available at AP Central, and is included in Appendix B.

Language Features and other Testable Topics

Tested in the AP CS A Exam	Notes	Not tested in the AP CS A Exam, but potentially relevant/useful
Comments `/* */`, `//`, and `/** */` Javadoc `@param` and `@return` comment tags		Javadoc tool
Primitive Types `int`, `double`, `boolean`		`char`, `byte`, `short`, `long`, `float`

Operators
Arithmetic: `+`, `−`, `*`, `/`, `%`
Increment/Decrement: `++`, `−−`
Assignment: `=`, `+=`, `−=`, `*=`, `/=`, `%=`
Relational: `==`, `!=`, `<`, `<=`, `>`, `>=`
Logical: `!`, `&&`, `||`
Numeric casts: `(int)`, `(double)`
String concatenation: `+`
	1, 2, 3, 4, 5	`&`, `|`, `^`
		`(char)`, `(float)`
		`StringBuilder`
		Shift: `<<`, `>>`, `>>>`
		Bitwise: `~`, `&`, `|`, `^`
		Conditional: `?:`

Object Comparison object identity `(==, !=)` vs. object equality `(equals)`, `String compareTo`		implementation of `equals` `Comparable`
Escape Sequences `\"`, `\\`, `\n` inside strings		`\'`, `\t`, `\unnnn`
Input / Output `System.out.print`, `System.out.println`	6	`Scanner`, `System.in`, `System.out`, `System.err`, `Stream` input/output, GUI input/output, parsing input: `Integer.parseInt`, `Double.parseDouble` formatting output: `System.out.printf`

Tested in the AP CS A Exam	Notes	Not tested in the AP CS A Exam, but potentially relevant/useful

Exceptions
`ArithmeticException,`
`NullPointerException,`
`IndexOutOfBoundsException,`
`ArrayIndexOutOfBoundsException,`
`IllegalArgumentException`
		`try/catch/finally`
		`throw, throws`
		`assert`

Arrays 1-dimensional arrays, 2-dimensional rectangular arrays, initializer list: `{ ... },` row-major order of 2-dimensional array elements	7, 8	`new` *type*`[]{ ... } ,` ragged arrays (non-rectangular), arrays with 3 or more dimensions

Control Statements
`if, if/else,`
`while, for,`
enhanced `for` (for-each),
`return`
		`switch,`
		`break, continue,`
		`do-while`

Variables parameter variables, local variables, `private` instance variables: visibility `(private)` `static` (class) variables: visibility `(public, private),` `final`		`final` parameter variables, `final` local variables, `final` instance variables

Methods
visibility `(public, private),`
`static, non-static,`
method signatures,
overloading, overriding,
parameter passing
	9, 10	visibility `(protected),`
		`public static void`
		`main(String[] args),`
		command line arguments,
		variable number of parameters,
		`final`

Constructors `super(), super(`*args*`)`	11, 12	default initialization of instance variables, initialization blocks, `this(`*args*`)`

Tested in the AP CS A Exam	Notes	Not tested in the AP CS A Exam, but potentially relevant/useful
Classes `new`, visibility `(public)`, accessor methods, modifier (mutator) methods Design/create/modify class. Create subclass of a superclass `(abstract,` non-`abstract).` Create class that implements an interface.	13, 14	`final`, visibility `(private, protected)`, nested classes, inner classes, enumerations
Interfaces Design/create/modify an interface.	13, 14	
Inheritance Understand inheritance hierarchies. Design/create/modify subclasses. Design/create/modify classes that implement interfaces.		
Packages `import` *packageName.className*		`import` *packageName*`.*` , `static` import, `package` *packageName* , class path
Miscellaneous OOP "is-a" and "has-a" relationships, `null`, `this`, `super.`*method(args)*	15, 16	`instanceof` *(class)* cast `this.`*var*, `this.`*method*`(`*args*`)`,
Standard Java Library `Object`, `Integer, Double`, `String`, `Math`, `List<E>, ArrayList<E>`	17, 18	`clone`, autoboxing, `Collection<E>`, `Arrays, Collections`

Notes

1. Students are expected to understand the operator precedence rules of the listed operators.
2. The increment/decrement operators `++` and `--` are part of the AP Java subset. These operators are used only for their side effect, not for their value. That is, the postfix form (for example, `x++`) is always used, and the operators are not used inside other expressions. For example, `arr[x++]` is not used.
3. Students need to understand the "short circuit" evaluation of the `&&` and `||` operators.
4. Students are expected to understand "truncation towards `0`" behavior as well as the fact that positive floating-point numbers can be rounded to the nearest integer as

   `(int)(x + 0.5)`, negative numbers as `(int)(x - 0.5)`.
5. String concatenation `+` is part of the AP Java subset. Students are expected to know that concatenation converts numbers to strings and invokes `toString` on objects.
6. User input is not included in the AP Java subset. There are many possible ways for supplying user input: e.g., by reading from a `Scanner`, reading from a stream (such as a file or a URL), or from a dialog box. There are advantages and disadvantages to the various approaches. The exam does not prescribe any one approach. Instead, if reading input is necessary, it will be indicated in a way similar to the following:

   `double x = /*` call to a method that reads a floating-point number */;

   or

   `double x = ...;    //` read user input
7. Both arrays of primitive types (e.g., `int[]`, `int[][]`) and arrays of objects (e.g., `Student[]`, `Student[][]`) are in the subset.
8. Students need to understand that 2-dimensional arrays are stored as arrays of arrays. For the purposes of the AP CS A Exam, students should assume that 2-dimensional arrays are rectangular (not ragged) and the elements are indexed in row-major order. For example, given the declaration

   ```
   int[][] m = {{1, 2, 3}, {4, 5, 6}};
   ```

   `m.length` is 2 (the number of rows), `m[0].length` is 3 (the number of columns), `m[r][c]` represents the element at row `r` and column `c`, and `m[r]` represents row `r` (e.g., `m[0]` is of type `int[]` and references the array `{1, 2, 3}`).

   Students are expected to be able to access a row of a 2-dimensional array, assign it to a 1-dimensional array reference, pass it as a parameter, and use loops (including for-each) to traverse the rows. However, students are not expected to analyze or implement code that replaces an entire row in a 2-dimensional array, such as

   ```
   int[][] m = {{1, 2, 3}, {4, 5, 6}};
   int[] a = {7, 8, 9};
   m[0] = a;    // Outside the Subset
   ```

9. The `main` method and command-line arguments are not included in the subset. In free-response questions, students are not expected to invoke programs. In the *AP Computer Science Labs*, program invocation with `main` may occur, but the `main` method will be kept very simple.
10. Students are required to understand when the use of `static` methods is appropriate. In the exam, `static` methods are always invoked through a class (explicitly or implicitly), never an object (i.e., *ClassName.staticMethod*`()` or *staticMethod*`()`, not *obj.staticMethod*`()`).
11. If a subclass constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass.
12. Students are expected to implement constructors that initialize all instance variables. Class constants are initialized with an initializer:

    ```
    public static final int MAX_SCORE = 5;
    ```

    The rules for default initialization (with `0, false` or `null`) are not included in the subset. Initializing instance variables with an initializer is not included in the subset. Initialization blocks are not included in the subset.
13. Students are expected to write interfaces or class declarations when given a general description of the interface or class.
14. Students are expected to extend classes and implement interfaces. Students are also expected to have knowledge of inheritance that includes understanding the concepts of method overriding and polymorphism. Students are expected to implement their own subclasses.

    Students are expected to read the definition of an abstract class and understand that the abstract methods need to be implemented in a subclass. Students are similarly expected to read the definition of an interface and understand that the abstract methods need to be implemented in an implementing class.
15. Students are expected to understand that conversion from a subclass reference to a superclass reference is legal and does not require a cast. Class casts (generally from `Object` to another class) are not included in the AP Java subset. Array type compatibility and casts between array types are not included in the subset.
16. The use of `this` is restricted to passing the implicit parameter in its entirety to another method (e.g., *obj.method*`(this)`) and to descriptions such as "the implicit parameter `this".` Students are not required to know the idiom "`this.`*var* = *var*", where *var* is both the name of an instance variable and a parameter variable.
17. The use of generic collection classes and interfaces is in the AP Java subset, but students need not implement generic classes or methods.
18. Students are expected to know a subset of the constants and methods of the listed Standard Java Library classes and interfaces. Those constants and methods are enumerated in the Java Quick Reference (Appendix B).

APPENDIX B

Exam Appendix — Java Quick Reference

Accessible methods from the Java library that may be included on the exam

class java.lang.Object

```
•  boolean equals(Object other)
•  String toString()
```

class java.lang.Integer

```
•  Integer(int value)
•  int intValue()
•  Integer.MIN_VALUE                  // minimum value represented by an int or Integer
•  Integer.MAX_VALUE                  // maximum value represented by an int or Integer
```

class java.lang.Double

```
•  Double(double value)
•  double doubleValue()
```

class java.lang.String

```
•  int length()
•  String substring(int from, int to) // returns the substring beginning at from
                                      // and ending at to-1
•  String substring(int from)         // returns substring(from, length())
•  int indexOf(String str)            // returns the index of the first occurrence of str;
                                      // returns -1 if not found
•  int compareTo(String other)        // returns a value < 0 if this is less than other
                                      // returns a value = 0 if this is equal to other
                                      // returns a value > 0 if this is greater than other
```

class java.lang.Math

```
•  static int abs(int x)
•  static double abs(double x)
•  static double pow(double base, double exponent)
•  static double sqrt(double x)
•  static double random()             // returns a double in the range [0.0, 1.0)
```

interface java.util.List<E>

```
•  int size()
•  boolean add(E obj)                 // appends obj to end of list; returns true
•  void add(int index, E obj)         // inserts obj at position index (0 ≤ index ≤ size),
                                      // moving elements at position index and higher
                                      // to the right (adds 1 to their indices) and adjusts size
•  E get(int index)
•  E set(int index, E obj)            // replaces the element at position index with obj
                                      // returns the element formerly at the specified position
•  E remove(int index)                // removes element from position index, moving elements
                                      // at position index + 1 and higher to the left
                                      // (subtracts 1 from their indices) and adjusts size
                                      // returns the element formerly at the specified position
```

class java.util.ArrayList<E> implements java.util.List<E>

APPENDIX C — SAMPLE SEARCH AND SORT ALGORITHMS

Sequential Search

The Sequential Search Algorithm below finds the index of a value in an array of integers as follows:

1. Traverse `elements` until `target` is located, or the end of `elements` is reached.
2. If `target` is located, return the index of `target` in `elements`;
   Otherwise return `-1`.

```
/**
 * Finds the index of a value in an array of integers.
 *
 * @param elements an array containing the items to be searched.
 * @param target the item to be found in elements.
 * @return an index of target in elements if found; -1 otherwise.
 */
public static int sequentialSearch(int[] elements, int target)
{
  for (int j = 0; j < elements.length; j++)
  {
    if (elements[j] == target)
    {
      return j;
    }
  }

  return -1;
}
```

Binary Search

The Binary Search Algorithm below finds the index of a value in an array of integers sorted in ascending order as follows:

1. Set `left` and `right` to the minimum and maximum indexes of `elements` respectively.
2. Loop until `target` is found, or `target` is determined not to be in `elements` by doing the following for each iteration:
    a. Set `middle` to the index of the middle item in `elements[left]` ... `elements[right]` inclusive.

    b. If `target` would have to be in `elements[left]` ... `elements[middle - 1]` inclusive, then set `right` to the maximum index for that range.

    c. Otherwise, if `target` would have to be in `elements[middle + 1]` ... `elements[right]` inclusive, then set `left` to the minimum index for that range.

    d. Otherwise, return `middle` because `target == elements[middle]`.
3. Return `-1` if `target` is not contained in `elements`.

```
/**
 *  Find the index of a value in an array of integers sorted in ascending order.
 *
 *  @param  elements  an array containing the items to be searched.
 *          Precondition: items in  elements  are sorted in ascending order.
 *  @param target  the item to be found in  elements.
 *  @return  an index of  target  in  elements  if  target  found;
 *          -1  otherwise.
 */
public static int binarySearch(int[] elements, int target)
{
  int left = 0;
  int right = elements.length - 1;

  while (left <= right)
  {
    int middle = (left + right) / 2;
    if (target < elements[middle])
    {
      right = middle - 1;
    }
    else if (target > elements[middle])
    {
      left = middle + 1;
    }
    else
    {
      return middle;
    }
  }

  return -1;
}
```

Selection Sort

The Selection Sort Algorithm below sorts an array of integers into ascending order as follows:

1. Loop from `j = 0` to `j = elements.length−2,` inclusive, completing `elements.length−1` passes.
2. In each pass, swap the item at index `j` with the minimum item in the rest of the array `(elements[j+1]` through `elements[elements.length−1]`).

At the end of each pass, items in `elements[0]` through `elements[j]` are in ascending order and each item in this sorted portion is at its final position in the array

```
/**
 * Sort an array of integers into ascending order.
 *
 * @param elements an array containing the items to be sorted.
 *
 * Postcondition: elements contains its original items and items in elements
 *                are sorted in ascending order.
 */
public static void selectionSort(int[] elements)
{
  for (int j = 0; j < elements.length − 1; j++)
  {
      int minIndex = j;
      for (int k = j + 1; k < elements.length; k++)
      {
        if (elements[k] < elements[minIndex])
        {
           minIndex = k;
        }
      }

      int temp = elements[j];
      elements[j] = elements[minIndex];
      elements[minIndex] = temp;
  }
}
```

Insertion Sort

The Insertion Sort Algorithm below sorts an array of integers into ascending order as follows:

1. Loop from `j = 1` to `j = elements.length−1` inclusive, completing `elements.length−1` passes.
2. In each pass, move the item at index `j` to its proper position in `elements[0]` to `elements[j]`:
   a. Copy item at index `j` to `temp`, creating a "vacant" element at index `j` (denoted by `possibleIndex`).
   b. Loop until the proper position to maintain ascending order is found for `temp`.
   c. In each inner loop iteration, move the "vacant" element one position lower in the array.
3. Copy `temp` into the identified correct position (at `possibleIndex`).

At the end of each pass, items at `elements[0]` through `elements[j]` are in ascending order.

```
/**
 * Sort an array of integers into ascending order.
 *
 * @param elements an array containing the items to be sorted.
 *
 * Postcondition: elements contains its original items and items in elements
 *                are sorted in ascending order.
 */
public static void insertionSort(int[] elements)
{
  for (int j = 1; j < elements.length; j++)
  {
    int temp = elements[j];
    int possibleIndex = j;
    while (possibleIndex > 0 && temp < elements[possibleIndex - 1])
    {
      elements[possibleIndex] = elements[possibleIndex - 1];
      possibleIndex--;
    }

    elements[possibleIndex] = temp;
  }
}
```

Merge Sort

The Merge Sort Algorithm below sorts an array of integers into ascending order as follows:

`mergeSort`

This top-level method creates the necessary temporary array and calls the `mergeSortHelper` recursive helper method.

`mergeSortHelper`

This recursive helper method uses the Merge Sort Algorithm to sort `elements[from]` ... `elements[to]` inclusive into ascending order:

1. If there is more than one item in this range,
   a. divide the items into two adjacent parts, and
   b. call `mergeSortHelper` to recursively sort each part, and
   c. call the `merge` helper method to merge the two parts into sorted order.
2. Otherwise, exit because these items are sorted.

`merge`

This helper method merges two adjacent array parts, each of which has been sorted into ascending order, into one array part that is sorted into ascending order:

1. As long as both array parts have at least one item that hasn't been copied, compare the first un-copied item in each part and copy the minimal item to the next position in `temp`.
2. Copy any remaining items of the first part to `temp`.
3. Copy any remaining items of the second part to `temp`.
4. Copy the items from `temp[from]` ... `temp[to]` inclusive to the respective locations in `elements`.

```
/**
 * Sort an array of integers into ascending order.
 *
 * @param elements an array containing the items to be sorted.
 *
 * Postcondition: elements contains its original items and items in elements
 *                are sorted in ascending order.
 */
public static void mergeSort(int[] elements)
{
  int n = elements.length;
  int[] temp = new int[n];
  mergeSortHelper(elements, 0, n — 1, temp);
}
```

```
/**
 * Sorts elements[from] ... elements[to] inclusive into ascending order.
 *
 * @param elements an array containing the items to be sorted.
 * @param from the beginning index of the items in elements to be sorted.
 * @param to the ending index of the items in elements to be sorted.
 * @param temp a temporary array to use during the merge process.
 *
 * Precondition:
 *     (elements.length == 0 or
 *      0 <= from <= to <= elements.length) and
 *     elements.length == temp.length
 * Postcondition: elements contains its original items and the items in elements
 *                [from] ... <= elements[to] are sorted in ascending order.
 */
private static void mergeSortHelper(int[] elements,
                                    int from, int to, int[] temp)
{
  if (from < to)
  {
    int middle = (from + to) / 2;
    mergeSortHelper(elements, from, middle, temp);
    mergeSortHelper(elements, middle + 1, to, temp);
    merge(elements, from, middle, to, temp);
  }
}
```

```
/**
 * Merges two adjacent array parts, each of which has been sorted into ascending
 * order, into one array part that is sorted into ascending order.
 *
 * @param elements an array containing the parts to be merged.
 * @param from the beginning index in elements of the first part.
 * @param mid the ending index in elements of the first part.
 *        mid+1 is the beginning index in elements of the second part.
 * @param to the ending index in elements of the second part.
 * @param temp a temporary array to use during the merge process.
 *
 * Precondition: 0 <= from <= mid <= to <= elements.length and
 *   elements[from] ... <= elements[mid] are sorted in ascending order and
 *   elements[mid + 1] ... <= elements[to] are sorted in ascending order and
 *   elements.length == temp.length
 * Postcondition: elements contains its original items and
 *   elements[from] ... <= elements[to] are sorted in ascending order and
 *   elements[0] ... elements[from - 1] are in original order and
 *   elements[to + 1] ... elements[elements.length - 1] are in original order.
 */
private static void merge(int[] elements,
                          int from, int mid, int to, int[] temp)
{
  int i = from;
  int j = mid + 1;
  int k = from;

  while (i <= mid && j <= to)
  {
    if (elements[i] < elements[j])
    {
      temp[k] = elements[i];
      i++;
    }
    else
    {
      temp[k] = elements[j];
      j++;
    }
    k++;
  }
  while (i <= mid)
  {
    temp[k] = elements[i];
    i++;
    k++;
  }

  while (j <= to)
  {
    temp[k] = elements[j];
    j++;
    k++;
  }

  for (k = from; k <= to; k++)
  {
    elements[k] = temp[k];
  }
}
```