# 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];
  }
}
```