Canonical Algorithm Implementations

AP Computer Science A

This handout presents the common textbook implementations of the core algorithms on the AP CSA radar: selection sort, insertion sort, merge sort (divide-and-conquer), and binary search. Bubble sort — no longer emphasized on the AP exam but still a common reference point — is included as an appendix. While the College Board does not prescribe exact code, these versions match what appears in the AP-style FRQs, Barron's, and most major AP textbooks.

1. Selection Sort

Idea: Repeatedly find the smallest remaining element and place it at the front of the unsorted region. After the i-th pass, the first i elements are in their final sorted positions.

Implementation

```
public static void selectionSort(int[] arr)
{
    for (int i = 0; i < arr.length - 1; i++)
    {
        int minIndex = i;
        for (int j = i + 1; j < arr.length; j++)
        {
            if (arr[j] < arr[minIndex])
            {
                minIndex = j;
            }
        }
        // swap arr[i] and arr[minIndex]
        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}
```

Key observations

- The outer loop runs n − 1 times; on the last pass, only one element remains, so it is guaranteed to be in place.
- minIndex is reset to i at the start of each pass, then updated whenever a smaller element is found.
- Exactly one swap per outer-loop iteration (at most n − 1 swaps total) — the fewest of any comparison sort.
- Not stable by default: swapping across equal elements can change their relative order.

2. Insertion Sort

Idea: Walk through the array from left to right; for each element, slide it leftward past any larger elements until it lands in its sorted position. Think of it like sorting a hand of playing cards as you pick them up one at a time.

Implementation

```
public static void insertionSort(int[] arr)
{
    for (int i = 1; i < arr.length; i++)
    {
        int current = arr[i];
        int j = i - 1;

        // shift larger elements one position to the right
        while (j >= 0 && arr[j] > current)
        {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = current;
    }
}
```

Key observations

- The outer loop starts at i = 1 because a single-element prefix is trivially sorted.
- current holds the value being inserted; the inner loop shifts larger elements one slot to the right rather than swapping, which is slightly more efficient than doing pairwise swaps.
- The inner loop's short-circuit (j >= 0 && arr[j] > current) stops as soon as the correct position is found — no full inner pass is needed.
- Best case O(n) on an already-sorted (or nearly-sorted) array, since the inner loop exits immediately each time. This is a real advantage over selection sort.
- Stable: equal elements are never moved past each other (note the strict > in the comparison).

3. Merge Sort

Idea: Recursively split the array in half, sort each half, and merge the two sorted halves back together. Merge sort is the standard example of divide-and-conquer in AP CSA.

Implementation

```
public static void mergeSort(int[] arr, int low, int high)
{
    if (low < high)
    {
        int mid = (low + high) / 2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}

public static void merge(int[] arr, int low, int mid, int high)
{
    int[] temp = new int[high - low + 1];
    int i = low;      // pointer into left half
    int j = mid + 1;  // pointer into right half
    int k = 0;        // pointer into temp

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

    // copy any remaining elements from the left half
    while (i <= mid)
    {
        temp[k] = arr[i];
        i++;
        k++;
    }

    // copy any remaining elements from the right half
    while (j <= high)
    {
        temp[k] = arr[j];
        j++;
        k++;
    }

    // copy temp back into arr
    for (int m = 0; m < temp.length; m++)
    {
        arr[low + m] = temp[m];
    }
}
```

Calling it

Invoke with the full range of the array:

```
mergeSort(arr, 0, arr.length - 1);
```

Key observations

- The recursion base case is implicit: when low >= high, the subarray has 0 or 1 elements and is already sorted, so the method simply returns.
- merge assumes both arr[low..mid] and arr[mid+1..high] are already sorted. It walks two pointers through the halves and writes to a temporary array in order.
- After the main while loop, at most one of the two halves still has elements; the two follow-up loops drain whichever half is left.
- Requires O(n) auxiliary space for the temp array — the price paid for guaranteed O(n log n) time.
- Stable when the comparison uses <= (as above) rather than <.

4. Binary Search

Idea: On a sorted array, repeatedly check the middle element and eliminate half the remaining range. Returns the index of the target, or −1 if it is not present.

Implementation (iterative)

```
public static int binarySearch(int[] arr, int target)
{
    int low = 0;
    int high = arr.length - 1;

    while (low <= high)
    {
        int mid = (low + high) / 2;

        if (arr[mid] == target)
        {
            return mid;
        }
        else if (arr[mid] < target)
        {
            low = mid + 1;
        }
        else
        {
            high = mid - 1;
        }
    }

    return -1; // target not found
}
```

Key observations

- Precondition: the array must be sorted in ascending order. Binary search on an unsorted array is undefined behavior in AP contexts.
- The loop condition is low <= high, not low < high — the search range is inclusive on both ends.
- Updates use mid + 1 and mid - 1, never just mid, to guarantee progress and avoid infinite loops.
- Each iteration halves the search space, giving O(log n) in the worst case.
- For an array of length n, the maximum number of iterations is $\lfloor \log_2 n \rfloor + 1$.

Complexity Summary

Algorithm	Best Case	Worst Case	Space	In-Place?
Selection Sort	O(n²)	O(n²)	O(1)	Yes
Insertion Sort	O(n)	O(n²)	O(1)	Yes
Bubble Sort	O(n²)*	O(n²)	O(1)	Yes
Merge Sort	O(n log n)	O(n log n)	O(n)	No
Binary Search	O(1)	O(log n)	O(1)	—

** Bubble sort reaches O(n) best case only when the early-exit flag optimization is included.*

Appendix: Bubble Sort

Bubble sort is no longer a focus of the AP CSA exam, but it still shows up often as a reference point when discussing sorting algorithms, so it is included here for completeness.

Idea: Repeatedly walk through the array and swap adjacent elements that are out of order. After each full pass, the largest remaining element has “bubbled up” to the end.

Implementation

```
public static void bubbleSort(int[] arr)
{
    for (int i = 0; i < arr.length - 1; i++)
    {
        for (int j = 0; j < arr.length - 1 - i; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                // swap arr[j] and arr[j + 1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Key observations

- The inner bound arr.length - 1 - i shrinks each pass because the last i elements are already in their final positions.
- Compares adjacent pairs (arr[j] and arr[j+1]) — this is the distinguishing feature versus selection sort.
- Stable: equal elements are never swapped past each other.
- An optional `swapped` flag can short-circuit the outer loop when a full pass makes no swaps, giving best-case O(n) on an already-sorted array. This optimization is often omitted in introductory materials.