Pseudocode:

```
int[] mergeSort(int[] arr)
        If (arr.length ≤ 1)
         Return arr // base case

        int mid ← arr.length / 2 // find midpoint

        int[] left ← new int[mid]
        int[] right ← new int[arr.length - mid]

        // copy left half
        For (int i = 0; i < mid; i++)
         left[i] ← arr[i]

        // copy right half
        For (int i = mid; i < arr.length; i++)
         right[i - mid] ← arr[i]

        // recursively sort and merge
        Return merge( mergeSort(left), mergeSort(right) )

//merge method
int[] merge(int[] a1, int[] a2)
        int[] result ← new int[a1.length + a2.length]
        int i ← 0, j ← 0, k ← 0 // counters

        // merge until one array is done
        While (i < a1.length AND j < a2.length)
         If (a1[i] ≤ a2[j])
          result[k] ← a1[i]
          i ← i + 1
         Else
          result[k] ← a2[j]
          j ← j + 1
         k ← k + 1

        // copy remaining from a1
        While (i < a1.length)
         result[k] ← a1[i]
         i ← i + 1
         k ← k + 1

        // copy remaining from a2
```

```
While (j < a2.length)
 result[k] ← a2[j]
 j ← j + 1
 k ← k + 1

Return result
```

Quick

```
void quickSort(int[] arr, int low, int high)
 If (low < high)
     int pivotIndex ← partition(arr, low, high) // partition step

     quickSort(arr, low, pivotIndex - 1)  // sort left part
     quickSort(arr, pivotIndex + 1, high) // sort right part


int partition(int[] arr, int low, int high)
    int pivot ← arr[high] // choose last element as pivot
    int i ← low - 1       // index of smaller element

    For (int j = low; j < high; j++)
     If (arr[j] ≤ pivot)
      i ← i + 1

      // swap arr[i] and arr[j]
      int temp ← arr[i]
      arr[i] ← arr[j]
      arr[j] ← temp

    // place pivot in correct position
    int temp ← arr[i + 1]
    arr[i + 1] ← arr[high]
    arr[high] ← temp

    Return i + 1 // new pivot index
```

Bucket

```
double[] bucketSort(double[] arr)
    If (arr is null OR arr.length = 0)
     Return arr
```

```
	n ← arr.length
	bucketCount ← floor(sqrt(n))

	// Create buckets (2D array), overallocate size n for each bucket
	double[][] buckets ← new double[bucketCount][n]
	int[] sizes ← new int[bucketCount]  // track count of elements in each bucket, initialized
	to 0

	// Distribute array elements into buckets
	For each num in arr
	 index ← floor(num * bucketCount)
	 If (index = bucketCount)
	  index ← index - 1  // handle case when num = 1.0

	 buckets[index][sizes[index]] ← num
	 sizes[index] ← sizes[index] + 1

	// Sort each bucket using insertion sort and merge into sorted array
	double[] sorted ← new double[n]
	idx ← 0

	For i from 0 to bucketCount - 1
	 bucketInsertionSort(buckets[i], sizes[i])  // sort only the filled part of bucket
		//This insertion sort will be coded seperately
	 For j from 0 to sizes[i] - 1
	  sorted[idx] ← buckets[i][j]
	  idx ← idx + 1

	Return sorted
```

Radix

```
void radixSort(int[] arr)
	If (arr is null OR arr.length = 0)
	 Return

	max ← getMax(arr)  // find the largest number in arr; code this seperately

	// Perform counting sort on each digit (1s, 10s, 100s, ...)
	exp ← 1
	While (max / exp > 0)
	 sortByDigit(arr, exp)
	 exp ← exp * 10
```

```
void sortByDigit(int[] arr, int exp)
        n ← arr.length
        int[] output ← new int[n]   // output array
        int[] count ← new int[10]   // for digits 0 to 9, initialized to 0

        // Count frequency of digits at current exponent place
        For i from 0 to n - 1
         digit ← (arr[i] / exp) mod 10
         count[digit] ← count[digit] + 1

        // Convert count to positions
        For i from 1 to 9
         count[i] ← count[i] + count[i - 1]

        // Build output array from right to left (to maintain stability)
        For i from n - 1 downto 0
         digit ← (arr[i] / exp) mod 10
         output[count[digit] - 1] ← arr[i]
         count[digit] ← count[digit] - 1

        // Copy output back to original array
        For i from 0 to n - 1
         arr[i] ← output[i]
```

a) Which algorithm performed best on each dataset? Explain why?
DS1: Quicksort performed best. Same time complexity as merge sort, but uses less space. Bucket sort is slow because of traversal of buckets utilizing insertion sort.
DS2: Radix sort performed best. Radix sort does not use comparison, allowing for a linear time complexity, whereas quick and merge sorts still perform comparisons.
DS3: Radix sort performed best. It does not rely on input order since it does not use comparisons. Merge also performs well since its logic is not dependent on how sorted the input is, but quick sort moves towards quadratic complexity since pivot is chosen is last element and input is almost completely sorted.
DS4: Radix sort does not use comparisons, so duplicates did not affect performance and so it performed best. Quick sort should also perform well despite using comparisons by effectively partitioning already sorted portions (did not perform as well due to me choosing last element as pivot; choosing middle as pivot would have resulted in efficient partitioning of values since many values will be the same). Merge sort treats this dataset as though it was completely unsorted and does a large number of merges.

b) How did input characteristics (e.g., range, order, duplicates) affect each algorithm's performance?
Merge:

Nearly sorted data (order) allows merge sort to take advantage of existing order by reducing unnecessary comparisons. Duplicates are processed as separate elements and does not take advantage of the fact that many of these input values are equal. Range does not affect performance, as inputs are directly compared based on their values, which takes constant time.

Quick:

Order negatively affects performance because pivot is chosen as last element (complexity almost turns quadratic) (in hindsight, I should have chosen the median as the pivot). Range does not affect performance (similar to merge sort above). Duplicates negatively affect performance due to pivot being chosen as last element rather than a middle element to give more effective partitioning.

Bucket:

Range may negatively affect performance, as inputs in this assignment were specified to be in the interval [0,1). If the range was much larger, buckets would have become uneven which would lead to slower running time. Duplicates may negatively affect performance as many duplicates would cause some buckets to be much larger and increase the time taken to sort buckets with many duplicates. Input order has little affect, but sorting within buckets affected by order of the bucket itself.

Radix:

Works best with integers in a fixed range since radix sort sorts digit by digit, and a uniform number of digits would yield more uniform comparisons. Order does not matter as input is processed digit by digit. Duplicates are handled well for the same reason.

c) Compare Merge Sort and Quick Sort on nearly sorted data. Which was better and why?

Merge sort: 2236300 ns ; QuickSort: 3220300 ns

Merge sort was better. Merge sort divides the array regardless of order/how sorted input is, so acts at worst the same as if input was not nearly sorted. Also since pivot was chosen as last element, time complexity of quick sort looked very close to O(n^2). A better implemented quick sort using middle/median element as pivot may have performed better than merge sort.

d) Under what scenarios are Bucket Sort or Radix Sort more effective than comparison-based sorts?

Bucket: Even distribution over known range (such as elements in range [0,1)). This keeps the buckets even in terms of size since they are evenly spaced so sorting within buckets is not too troublesome. Bucket sort is also good for when the number of elements is both evenly distributed and the number of elements is not vastly greater than the number of buckets.

Radix: Fixed range of values, as radix sort sorts digit by digit and as such duplicate values are also handled very well in comparison to other sorting algorithms.

e) Discuss the space complexity of each algorithm.

Merge

O(n)

Due to temporary subarrays used in merging.

Quick

O(log(n))

In place, and does not need external arrays, but good pivots can lead to O(log(n)) complexity in recursion stack. Bad pivot choice may lead to O(n) stack height.

Bucket

O(n + k)

Space for array as well as k buckets.

Radix

O(n + k)

Uses space for array as well as temporary arrays for each digit iteration.

OUTPUTS

```
C:\Users\tonyy\2025Summer\CSI2110\Assignments\Assignment3>java Sorts
DS1: Random floating-point numbers in the range [0, 1)
Sorting with merge
Merge sort took 4322000 ns
Sorting with quick
quick sort took 3079900 ns
Sorting with buckets
bucket sort took 8338100 ns



DS2: Random non-negative integers
Sorting with merge
Merge sort took 5075000 ns
Sorting with quick
quick sort took 4882500 ns
Sorting with radix
radix sort took 3375000 ns



DS3: A nearly sorted list (e.g., 90% sorted + 10% random noise)
Sorting with merge
Merge sort took 2236300 ns
Sorting with quick
quick sort took 3220300 ns
Sorting with radix
radix sort took 1675300 ns



DS4: A list with many duplicate values
Sorting with merge
Merge sort took 2425600 ns
Sorting with quick
quick sort took 1246900 ns
Sorting with radix
radix sort took 1052700 ns

C:\Users\tonyy\2025Summer\CSI2110\Assignments\Assignment3>
```