Sort type	Pseudo Code	Java Code

Bubble Sort
 	
....
Time Complexity: O(n^2)
Space Complexity: O(1)
....
Swap adjacent elements after comparison.
Largest number "floats" to last index.
	bubbleSort(int arr[]):
	    for i = 0 to (arr.length-1)
	        do: for j = 0 to (arr.length-i-1)
	            do: if arr[j] > arr[j+1]
	                then: temp <-- arr[j]
	                arr[j] <-- arr[j+1]
	                arr[j+1] <-- temp
		void bubbleSort(int arr[]){
		    int i, j, n = arr.length;
		    for (i = 0; i < n-1; i++)
		    // Last i elements are already in place
		        for (j = 0; j < n-i-1; j++)
		            if (arr[j] > arr[j+1]){
		                int temp = arr[j];
		                arr[j] = arr[j+1];
		                arr[j+1] = temp;
		            }
		}

Selection Sort
 	
....
Time Complexity: O(n^2)
Space Complexity: O(1)
....
Find minimum and put in the start.
	selectionSort(int arr[]):
	    for i = 0 to (arr.length-1)
	        do: min_id <-- i
	        for j = (i+1) to (arr.length)
	            do: if (arr[j] < arr[min_id])
	                then: min_id <-- j
	        temp <-- arr[i]
	        arr[i] <-- arr[min_id]
	        arr[min_id] <-- temp
		void selectionSort(int arr[]){
		    int i, j, min_id, n = arr.length;
		    // One by one move boundary of
		    // unsorted subarray
		    for (i = 0; i < n-1; i++){
		        // Find the minimum element
		        // in unsorted array
		        min_id = i;
		        for (j = i+1; j < n; j++)
		            if (arr[j] < arr[min_id])
		                min_id = j;
		        // Swap the found minimum element
		        // with the first element
		        int temp = arr[i];
		        arr[i] = arr[min_id];
		        arr[min_id] = temp;
		    }
		}

Insertion Sort

....
Time Complexity: O(n^2)
Space Complexity: O(1)
....
Pick values from the unsorted part and place at the correct position in the sorted part.

```
insertionSort(int arr[]):
    for i = 1 to (arr.length)
        do: key <-- arr[i]
        j <-- (i - 1)
        while(j >= 0 and arr[j] > key)
            do: arr[j+1] <-- arr[j]
            j <-- j - 1
        arr[j+1] <-- key
```

```
void insertionSort(int arr[]){
    int i, key, j, n = arr.length;
    for (i = 1; i < n; i++){
        key = arr[i];
        j = i - 1;
        // Move elements of arr[0..i-1], that are
        // greater than key, to one position ahead
        // of their current position
        while (j >= 0 && arr[j] > key){
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Merge Sort

....
Time Complexity: O(n lg(n))
Space Complexity: O(n)
....
Divide and Conquer
Divide input array in two halves, call sort function for the two halves and then

```
merge(A[],B[]):
    n1 = A.length, n2 = B.length,
    Create C[1 ... (n1+n2)]
    i = 0, j = 0, k = 0
    while i < n1 and j < n2
        do: if A[i] < B [j]
            then: C[k] = A[i]
            i = i+1
        else
            do: C[k] = B[j]
            j = j+1
        k = k+1
    while i < n1
        do: C[k] = A[i]
        k = k+1
        i = i+1
    while j < n2
        do: C[k] = B[j]
```

```
int[] merge(int[] a, int[] b){
    int p = a.length, q = b.length;
    int c[] = new int[p + q];
    int i = 0, j = 0, k = 0;
    while(i<p && j<q){
        if(a[i]<b[j]) c[k++] = a[i++];
        else c[k++] = b[j++];
    }
    while(i<p) c[k++] = a[i++];
    while(j<q) c[k++] = b[j++];
    return c;
}
int[] sort(int[] arr){
    int l = 0, r = arr.length-1;
    if(l < r){
        int m = (l + r) / 2; // middle point
        int n1 = m - l + 1, n2 = r - m;
        // Create temp arrays
```

for the two halves and then merge the two sorted halves.
Division runs until each sub array is of length 1, then merge takes place
	do: C[k] = B[j]
	        k = k+1
	        j = j+1
	    return C
	sort(Arr[]):
	    if Arr.length = 1
	        then return
	        else
	        Create L[] with A[0 to (n/2)]
	        Create R[] with A[(n/2+1) to n]
	        L = sort(L)
	        R = sort(R)
	        A = merge(L,R)
		// Create temp arrays
		        int[] L = new int[n1], R = new int[n2];
		        // Copy data to temp arrays
		        for (int i=0; i<n1; ++i) L[i] = arr[l+i];
		        for (int j=0; j<n2; ++j) R[j] = arr[m+1+j];
		        // Sort first and second halves
		        L = sort(L);
		        R = sort(R);
		        // Merge the sorted halves
		        arr = merge(L,R);
		    } // l==r
		    return arr;
		}

Heap Sort
....
Time Complexity: O(n lg(n))
Space Complexity: O(1)
....
Based on Binary Heap data structure.
Find the maximum element and place the maximum element at the end.
Very Similar to Selection Sort.
	build_max_heap(A)
	    heap_size[A] = length[A]
	    for i = (length[A]/2) downto 1
	        do max_heapify(A, i)
	max_heapify(A, i)
	    l = left(i)
	    r = right(i)
	    if l <= heap_size[A] and A[l] > A[i]
	        then: max = l
	        else: max = i
	    if r <= heap_size[A] and A[r] > A[max]
	        then: max = r
	    if max != i
	        then: exchange A[i] with A[max]
	        max_heapify(A, max)
	heapsort(A[])
	    build_max_heap(A)
	    for i = (A.length) downto 2
	        do: exchange A[1] with A[i]
	        heap_size[A] = heap_size[A] - 1
	        max_heapify(A, 1)
		This link has full heap class and implementation:
		https://drive.google.com/file/d/1hAN94NQ7zmZmfqK7BEwZvYf7oWkdjQQX/view?usp=sharing
		
		It can simply be used by keeping in the same package as driver class.

Quick Sort
 	
....
Time Complexity:
Average: O(n lg(n))
Worst: O(n^2)
Space Complexity: ~~~
....
Divide and Conquer
Pick an element as pivot and partition the array around the pivot with lesser elements on one side and greater elements on the other
The key process is partition()
In Randomized_QuickSort, pivots are just picked randomly
	<pre>partition(A, p, q) >>>>> A[p ... q]
	    x = A[p]
	    i = p
	    for j = p + 1 to q
	        do if A[j] <= x
	            then i = i + 1
	            exchange A[i] <> A[j]
	    exchange A[p] <> A[i]
	    return i
	quicksort(A, p, r) >>>>> A[p ... r]
	    if p < r
	        then q = partition(A, p, r)
	        quicksort(A, p, q - 1)
	        quicksort(A, q + 1, r)</pre>
		<pre>int partition(int[] A, int p, int q) {
		    int x = A[p]; // pivot
		    int i = p; // first index
		    for (int j = p+1; j <= q; j++) {
		        if(A[j]<=x){
		            i = i+1;
		            int t = A[i];
		            A[i] = A[j];
		            A[j] = t;
		        }
		    }
		    // swap first to put in between
		    // lesser and greater parts
		    int t = A[i];
		    A[i] = A[p];
		    A[p] = t;
		    return i;
		}
		int[] quick_sort(int[] A, int p, int r) {
		    if(p<r){
		        int q = partition(A,p,r);
		        quick_sort(A,p,q-1);
		        quick_sort(A,q+1, r);
		    }
		    return A;
		}</pre>