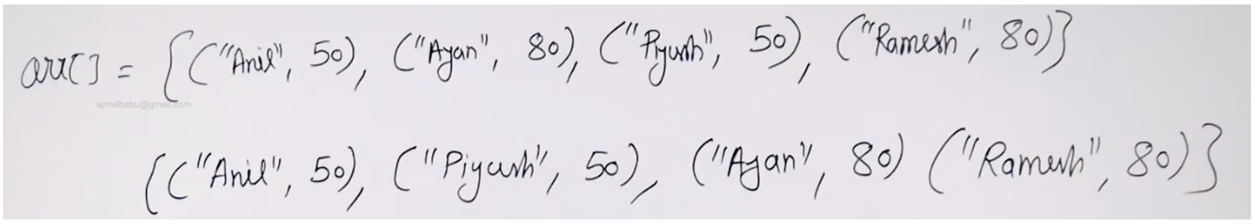
Leet code, making dynamic programming simple:
https://leetcode.com/discuss/study-guide/1490172/dynamic-programming-is-simple

https://www.educative.io/courses/grokking-the-coding-interview/xl0EIGxR6Bq

Log in with ajmalbabu@gmail.com
**Course name:** https://practice.geeksforgeeks.org/batch/cip-1#

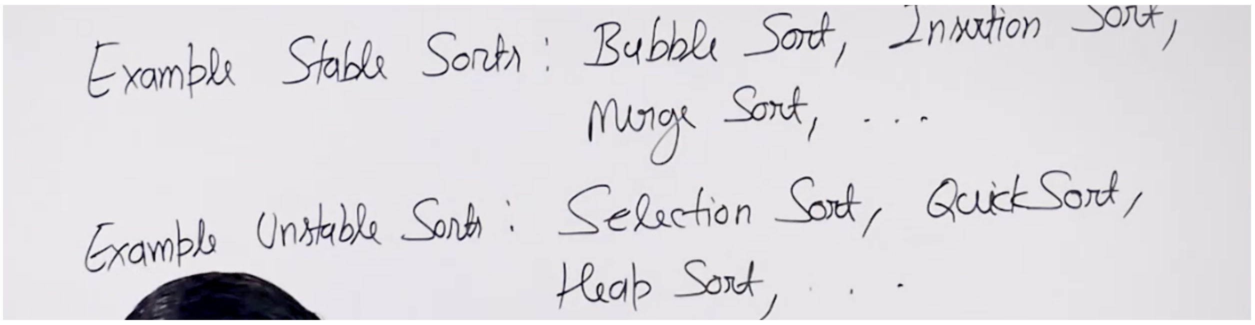
**Sorting:**

**Stable sorting:** Ensure that people who have the same marks appear in the same order in the original array as below. Let us say we are sorting by marks. Stability is important when there are multiple fields in the original array, but not applicable if all the elements are integers in the original array. Unstable can produce different output from the original



arr[] = {("Anil", 50), ("Ayan", 80), ("Piyush", 50), ("Ramesh", 80)}

{("Anil", 50), ("Piyush", 50), ("Ayan", 80) ("Ramesh", 80)}

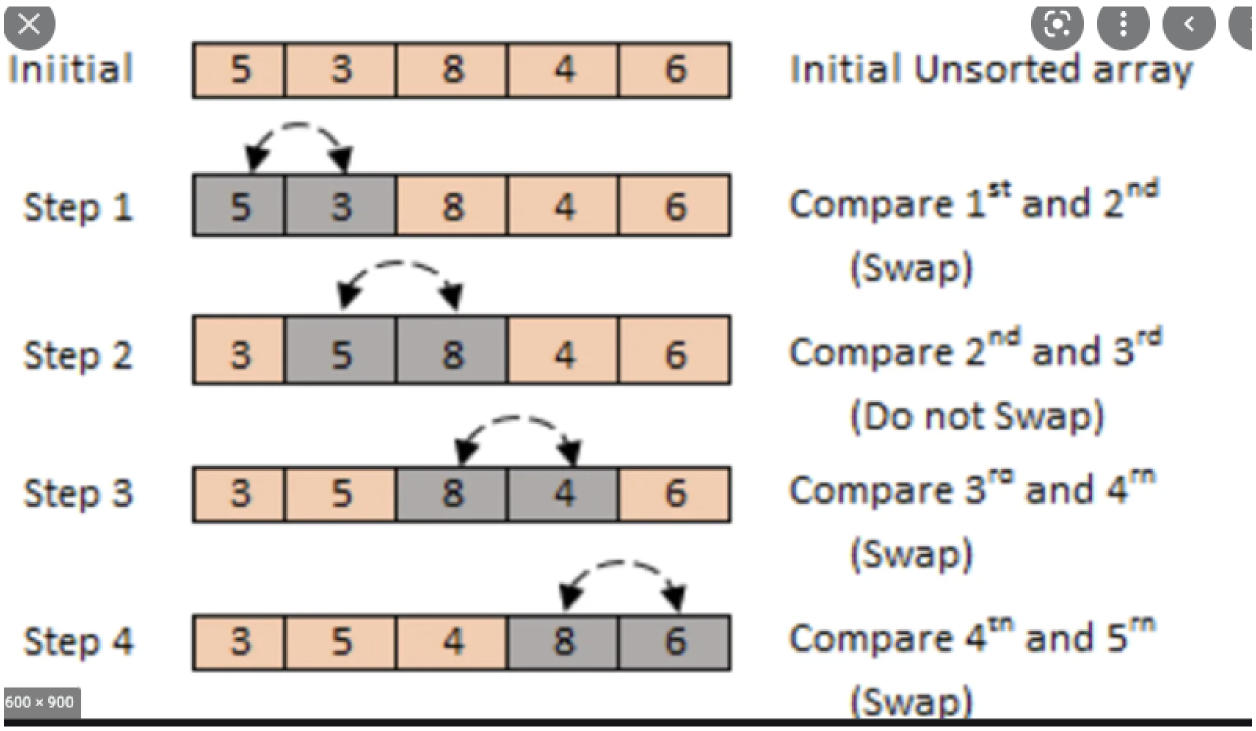


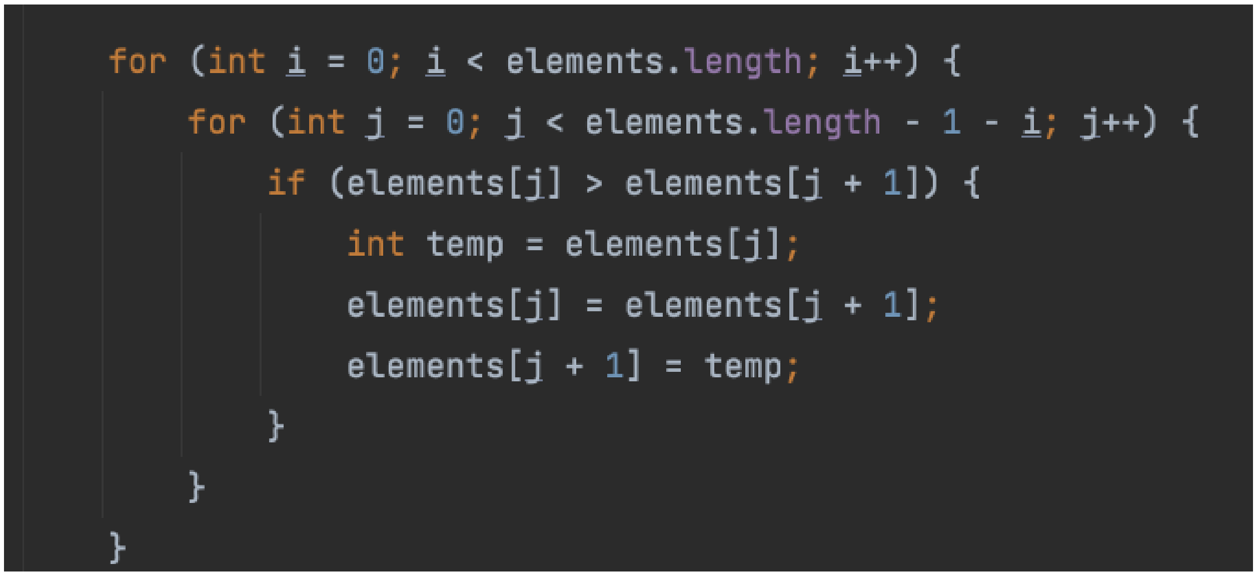
Example Stable Sorts : Bubble Sort, Insertion Sort,
Merge Sort, . . .

Example Unstable Sorts : Selection Sort, QuickSort,
Heap Sort, . . .

1. Bubble Sort

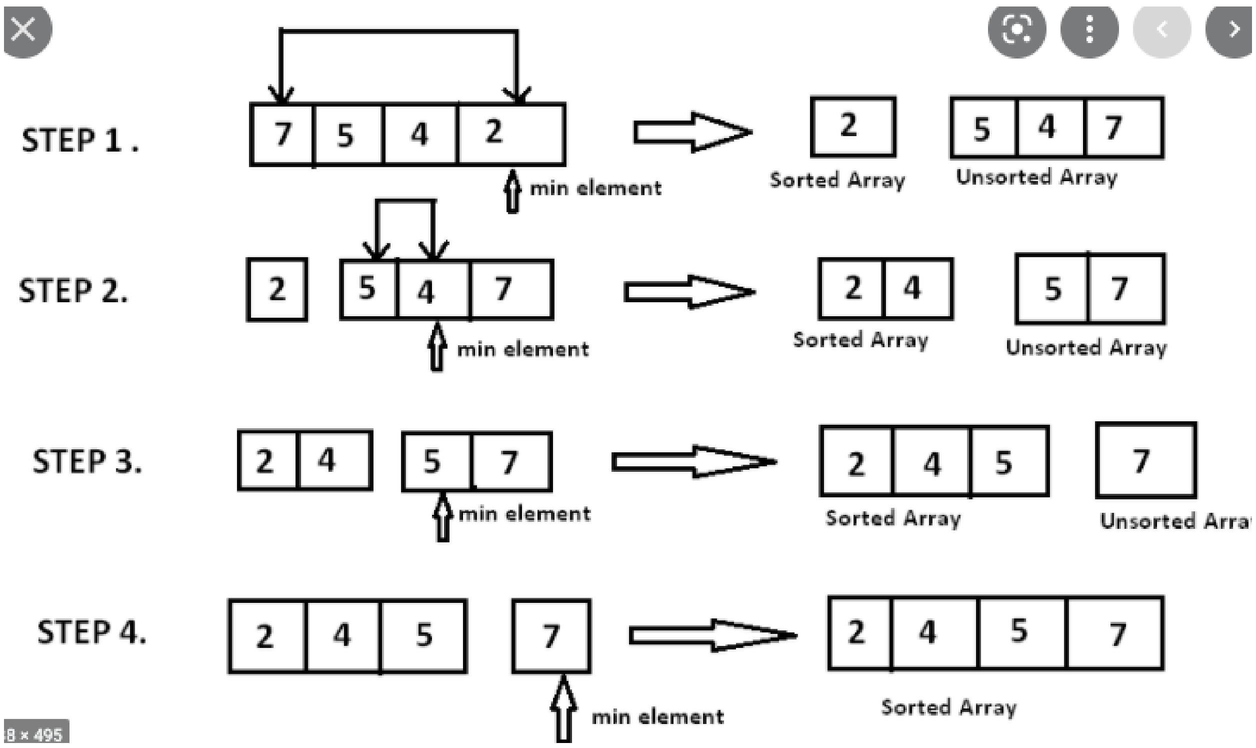
O(N^2)

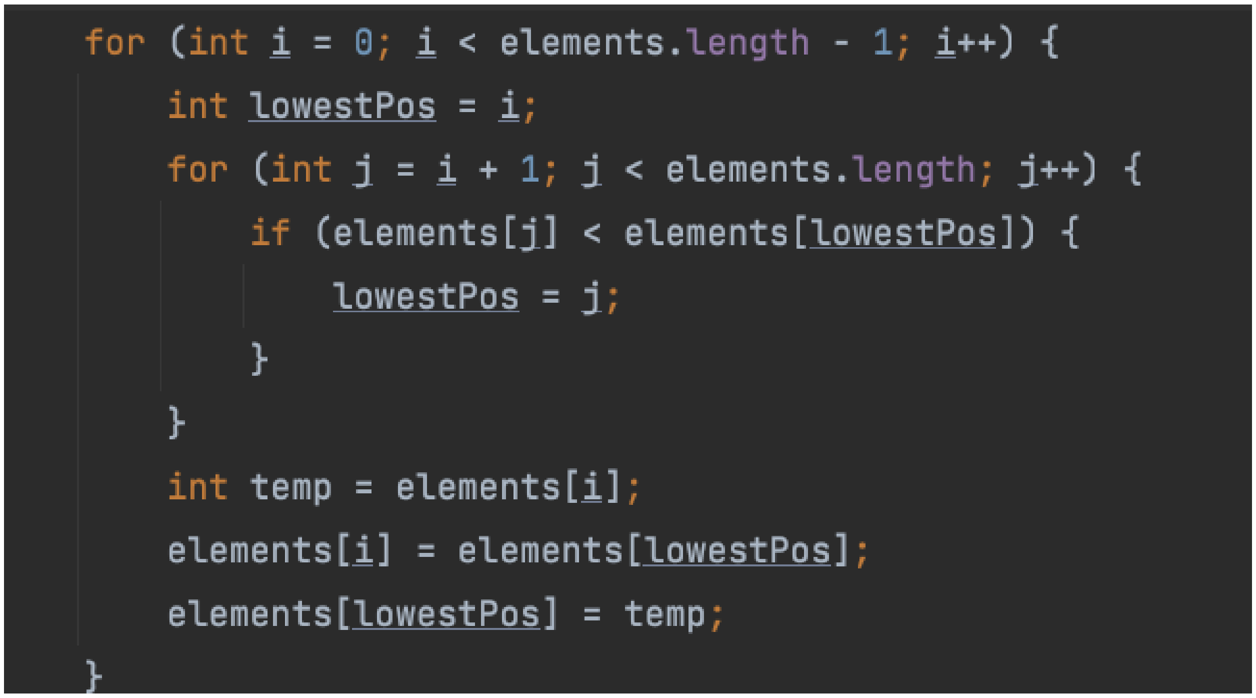

Iniitial
Initial Unsorted array
Step 1
Compare 1st and 2nd (Swap)
Step 2
Compare 2nd and 3rd (Do not Swap)
Step 3
Compare 3rd and 4th (Swap)
Step 4
Compare 4th and 5th (Swap)
600 × 900



```
for (int i = 0; i < elements.length; i++) {
    for (int j = 0; j < elements.length - 1 - i; j++) {
        if (elements[j] > elements[j + 1]) {
            int temp = elements[j];
            elements[j] = elements[j + 1];
            elements[j + 1] = temp;
        }
    }
}
```

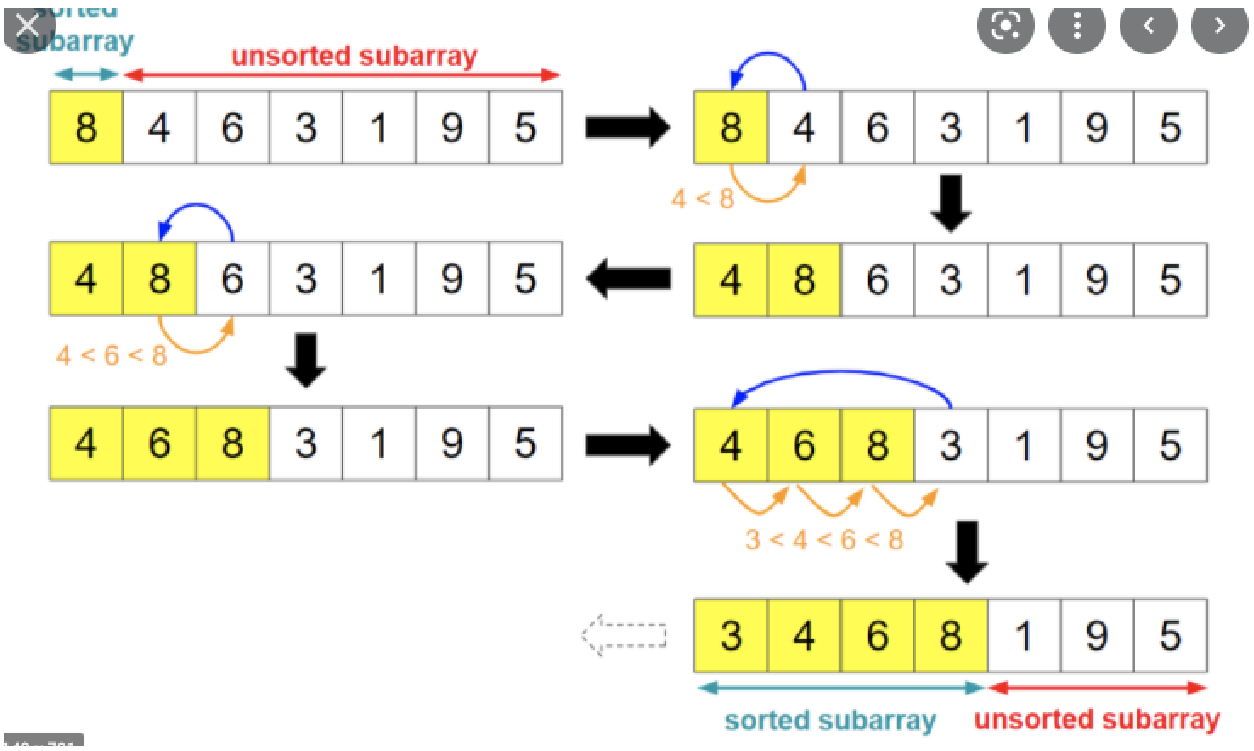
2. Selection Sort O(N^2)

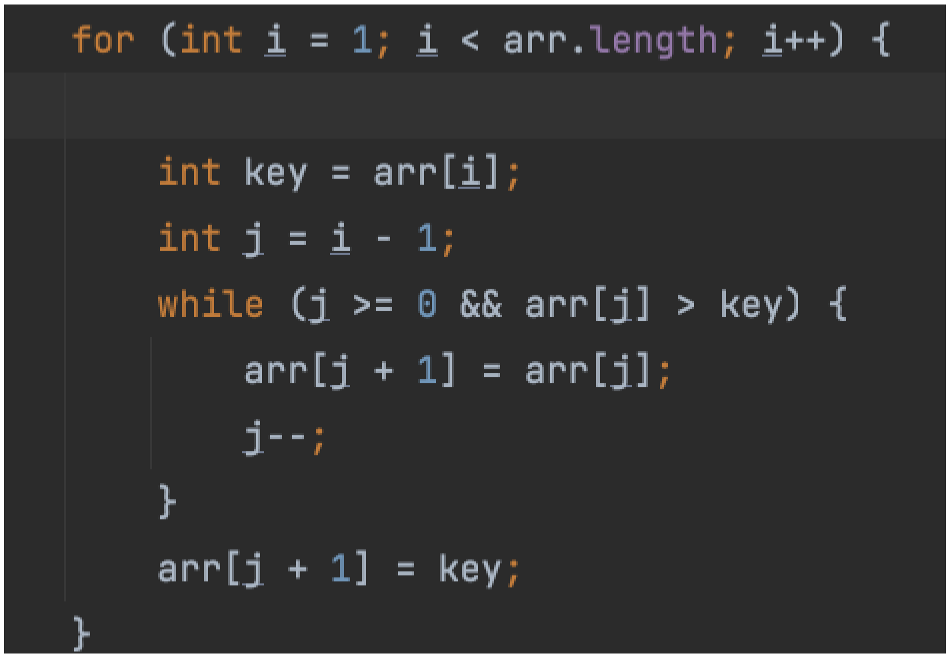




```
for (int i = 0; i < elements.length - 1; i++) {
    int lowestPos = i;
    for (int j = i + 1; j < elements.length; j++) {
        if (elements[j] < elements[lowestPos]) {
            lowestPos = j;
        }
    }
    int temp = elements[i];
    elements[i] = elements[lowestPos];
    elements[lowestPos] = temp;
}
```

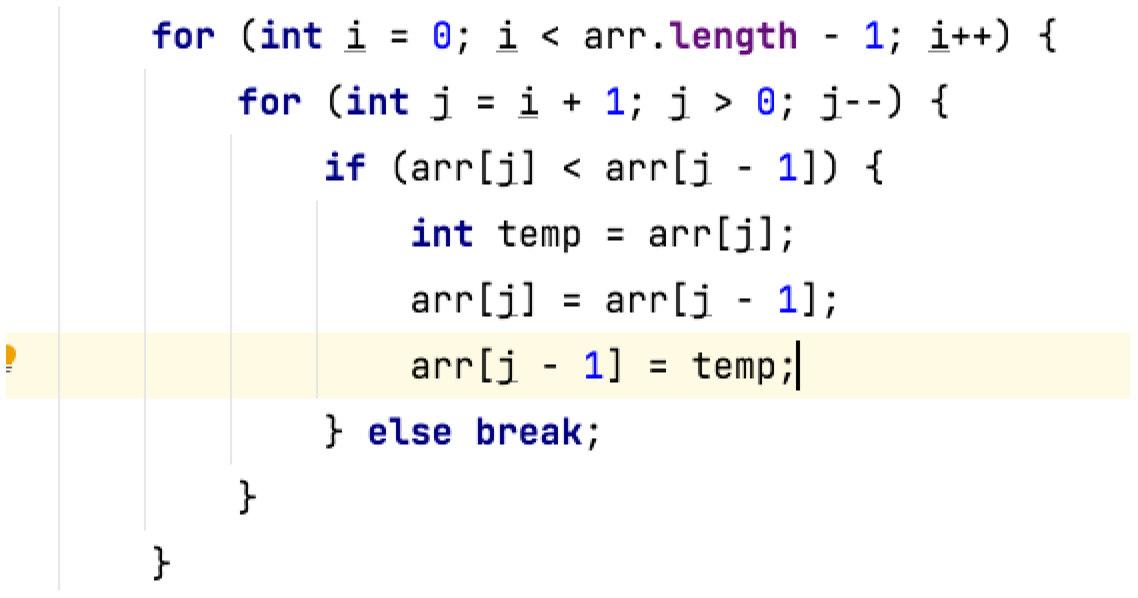
3. **Insertion sort O(N^2)**





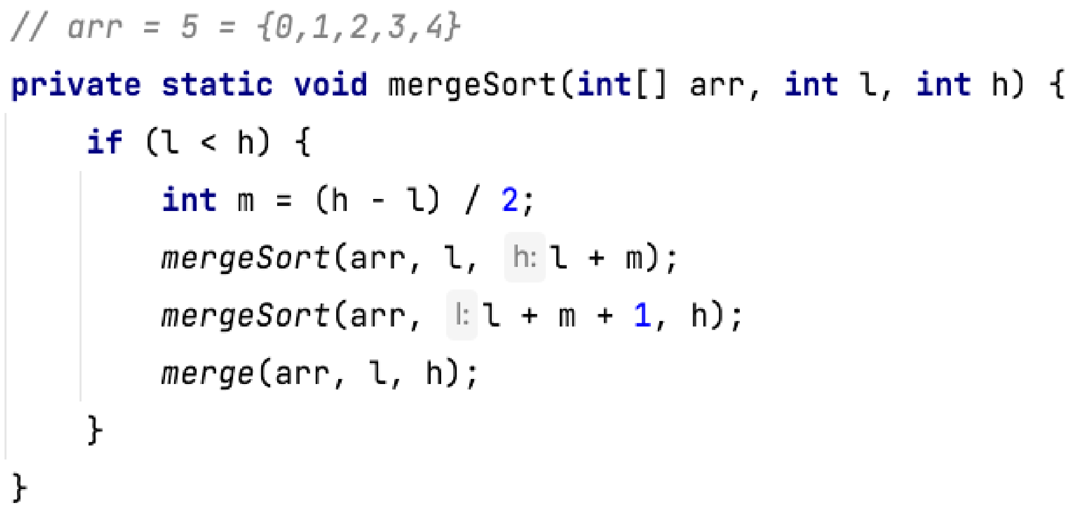
```
for (int i = 1; i < arr.length; i++) {

    int key = arr[i];
    int j = i - 1;
    while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j];
        j--;
    }
    arr[j + 1] = key;
}
```

**Better version from me below**



```
for (int i = 0; i < arr.length - 1; i++) {
    for (int j = i + 1; j > 0; j--) {
        if (arr[j] < arr[j - 1]) {
            int temp = arr[j];
            arr[j] = arr[j - 1];
            arr[j - 1] = temp;
        } else break;
    }
}
```

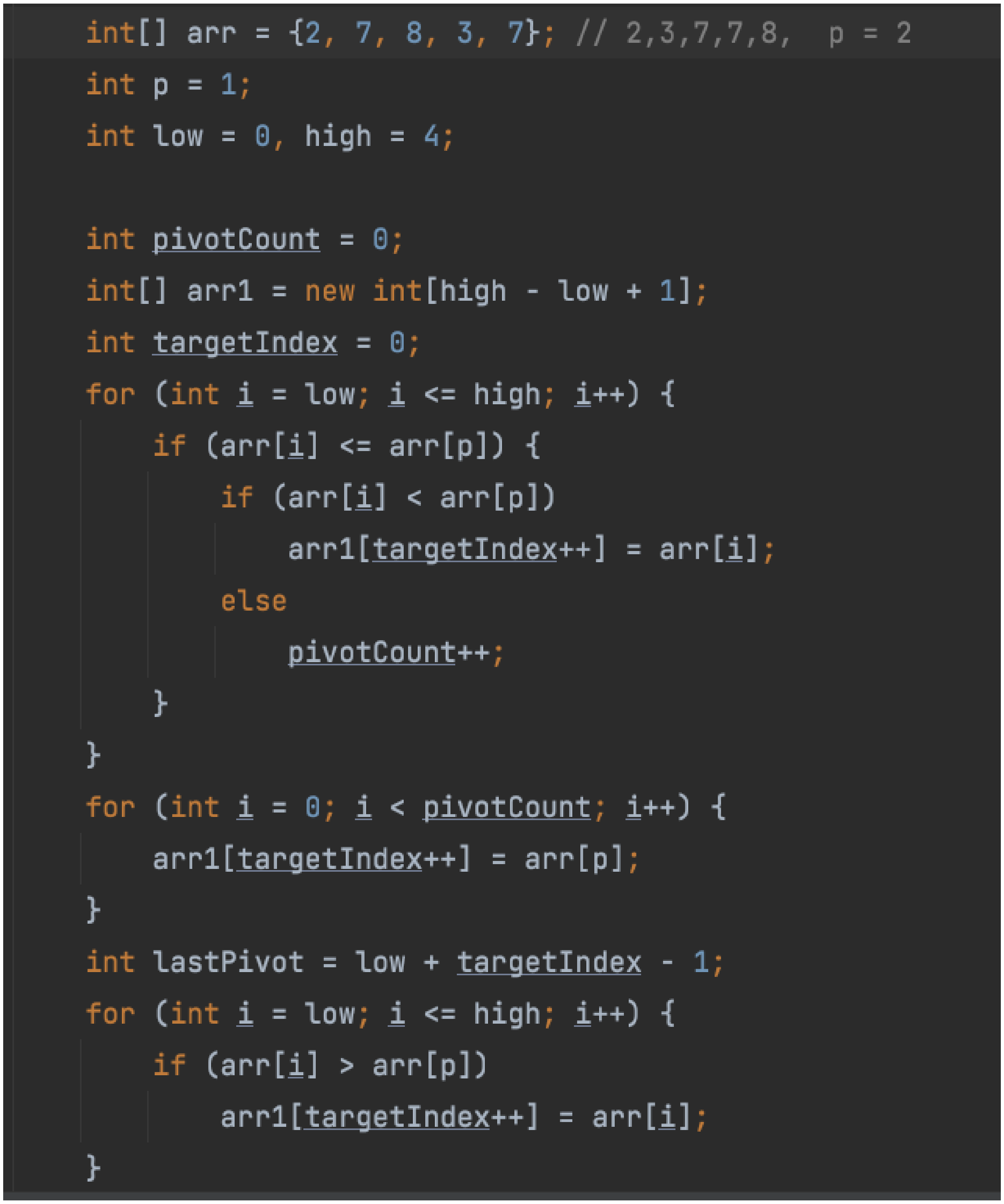
4. **Merge Sort  O(N log N) for worst-case and space complexity = O(N)**



```
// arr = 5 = {0,1,2,3,4}
private static void mergeSort(int[] arr, int l, int h) {
    if (l < h) {
        int m = (h - l) / 2;
        mergeSort(arr, l, h: l + m);
        mergeSort(arr, l: l + m + 1, h);
        merge(arr, l, h);
    }
}
```

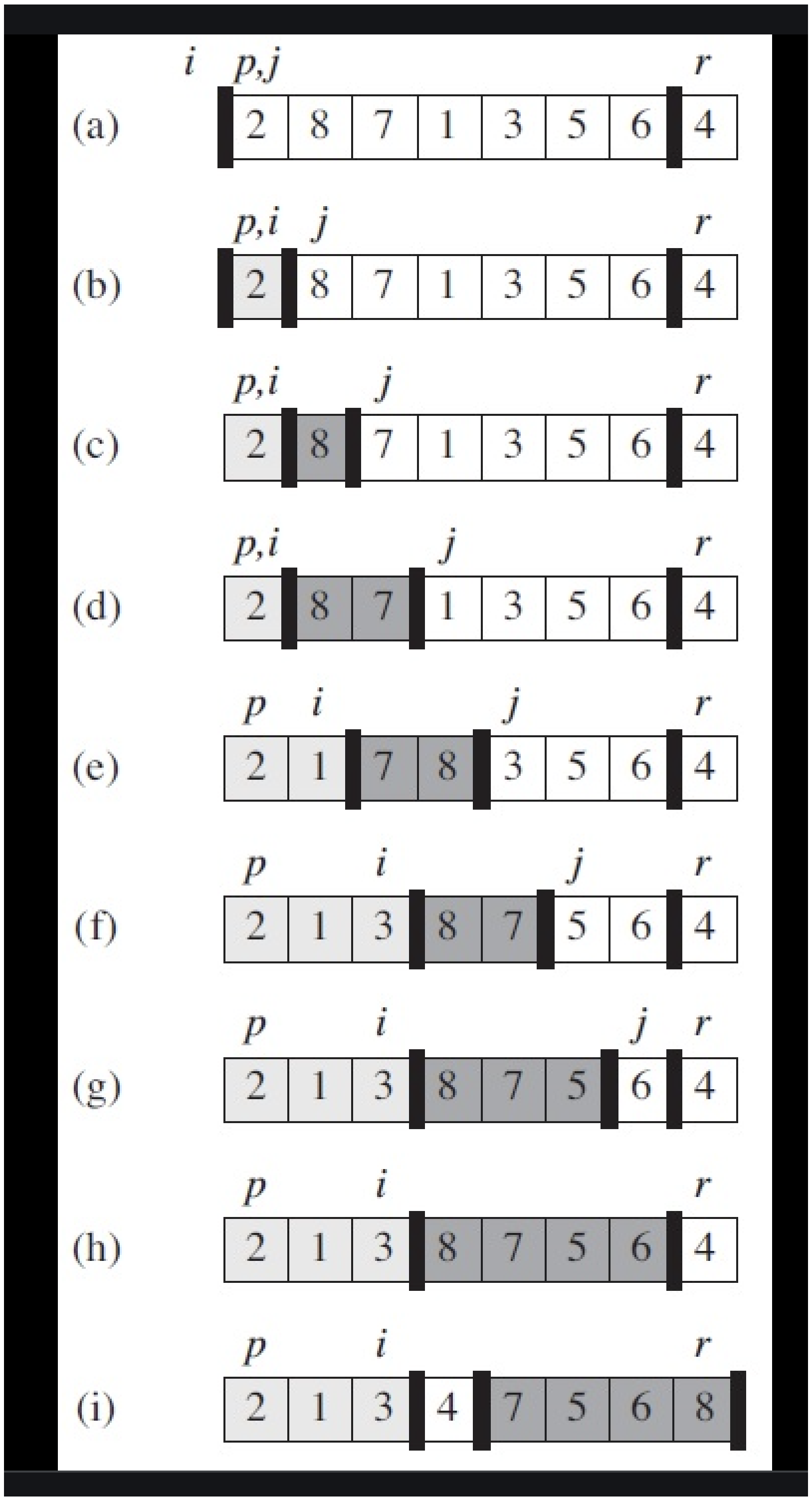
```java
private static void merge(int[] arr, int l, int h) {
    int m = (h - l) / 2;
    int[] a1 = new int[m + 1];
    int[] a2 = new int[h - l - m]; // or  (h - l + 1 - ( m + 1 ))

    int i = 0, a1Index = 0, a2Index = 0;

    for (i = l; i <= l + m; i++)
        a1[a1Index++] = arr[i];
    for (i = l + m + 1; i <= h; i++)
        a2[a2Index++] = arr[i];

    i = l;
    a1Index = 0;
    a2Index = 0;

    while (a1Index < a1.length && a2Index < a2.length) {
        if (a1[a1Index] == a2[a2Index]) {
            arr[i++] = a1[a1Index++];
            arr[i++] = a2[a2Index++];
        } else if (a1[a1Index] < a2[a2Index]) {
            arr[i++] = a1[a1Index++];
        } else {
            arr[i++] = a2[a2Index++];
        }
    }

    for (; a1Index < a1.length; a1Index++)
        arr[i++] = a1[a1Index];
    for (; a2Index < a2.length; a2Index++)
        arr[i++] = a2[a2Index];
}
```

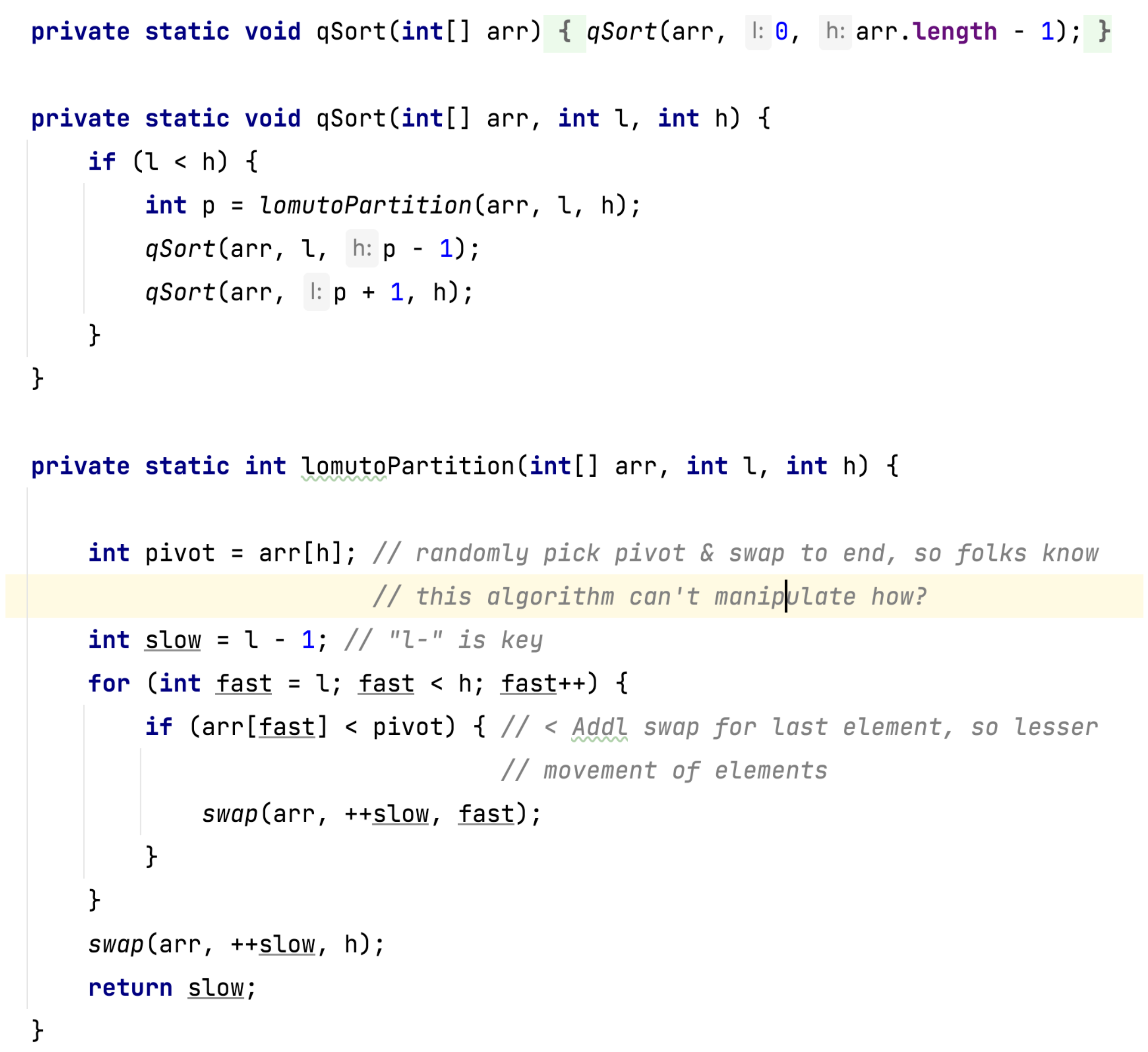
1. **Naive partition O(N) space is O(1)**



```
int[] arr = {2, 7, 8, 3, 7}; // 2,3,7,7,8,  p = 2
int p = 1;
int low = 0, high = 4;

int pivotCount = 0;
int[] arr1 = new int[high - low + 1];
int targetIndex = 0;
for (int i = low; i <= high; i++) {
    if (arr[i] <= arr[p]) {
        if (arr[i] < arr[p])
            arr1[targetIndex++] = arr[i];
        else
            pivotCount++;
    }
}
for (int i = 0; i < pivotCount; i++) {
    arr1[targetIndex++] = arr[p];
}
int lastPivot = low + targetIndex - 1;
for (int i = low; i <= high; i++) {
    if (arr[i] > arr[p])
        arr1[targetIndex++] = arr[i];
}
```

2. **Lomuto partition O(N * log N) and space O(1), time complexity worst-case O(N^2), and the average case is O(n log n). See a more detailed explanation in Hoare below**


(a) i p,j r 2 8 7 1 3 5 6 4
(b) p,i j r 2 8 7 1 3 5 6 4
(c) p,i j r 2 8 7 1 3 5 6 4
(d) p,i j r 2 8 7 1 3 5 6 4
(e) p i j r 2 1 7 8 3 5 6 4
(f) p i j r 2 1 3 8 7 5 6 4
(g) p i j r 2 1 3 8 7 5 6 4
(h) p i r 2 1 3 8 7 5 6 4
(i) p i r 2 1 3 4 7 5 6 8

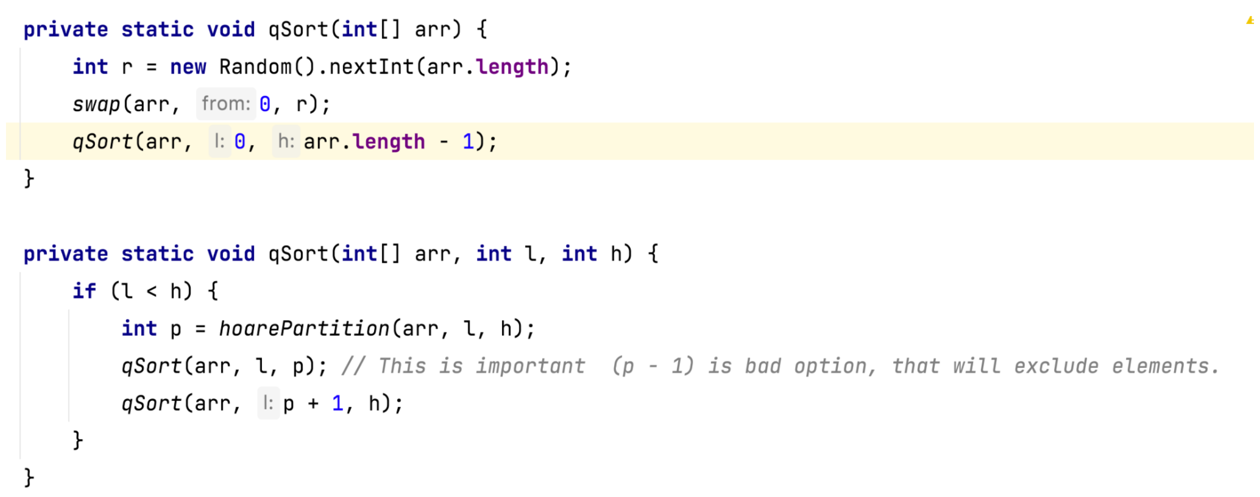


```
private static void qSort(int[] arr) { qSort(arr, l: 0, h: arr.length - 1); }

private static void qSort(int[] arr, int l, int h) {
    if (l < h) {
        int p = lomutoPartition(arr, l, h);
        qSort(arr, l, h: p - 1);
        qSort(arr, l: p + 1, h);
    }
}

private static int lomutoPartition(int[] arr, int l, int h) {

    int pivot = arr[h]; // randomly pick pivot & swap to end, so folks know
                        // this algorithm can't manipulate how?
    int slow = l - 1; // "l-" is key
    for (int fast = l; fast < h; fast++) {
        if (arr[fast] < pivot) { // < Addl swap for last element, so lesser
                                 // movement of elements
            swap(arr, ++slow, fast);
        }
    }
    swap(arr, ++slow, h);
    return slow;
}
```

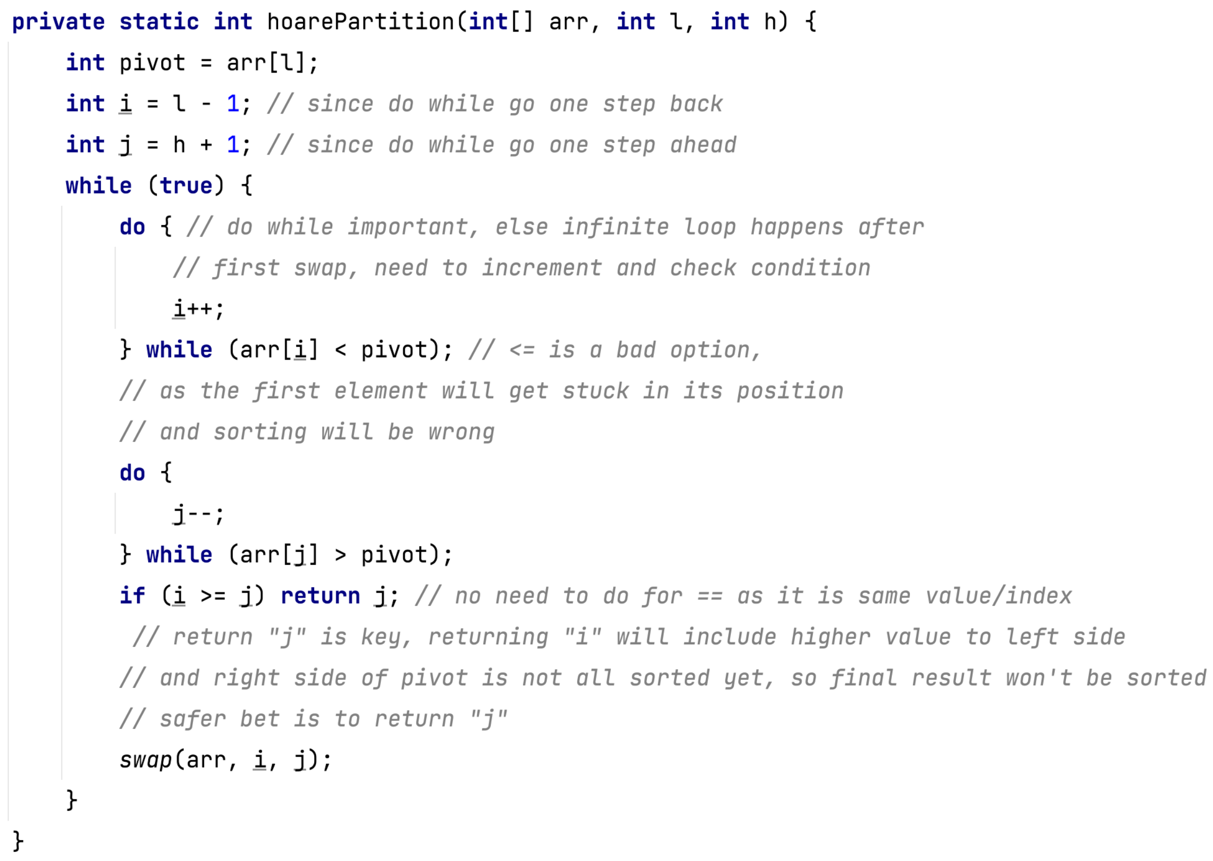
**fast < h** is a key idea if it is **fast <= h** there will be too many movements, the only drawback for **fast < h** is one additional code for the last element (pivot) swap

***HOARE***

3. **Hoare partition average case O(N logN), space complexity O(1)**
4. **worst case (for fully sorted array happens for every element) and time complexity will be N^2 (n + n-1 + n-2 + n-3 …1), space complexity O(1)**
5. **average complexity when 10% tree happens is = N + 9/10N + 9/(9/10)N + …. 1 = N *log (n) base 9/10 = N log (n), space complexity O(1)**

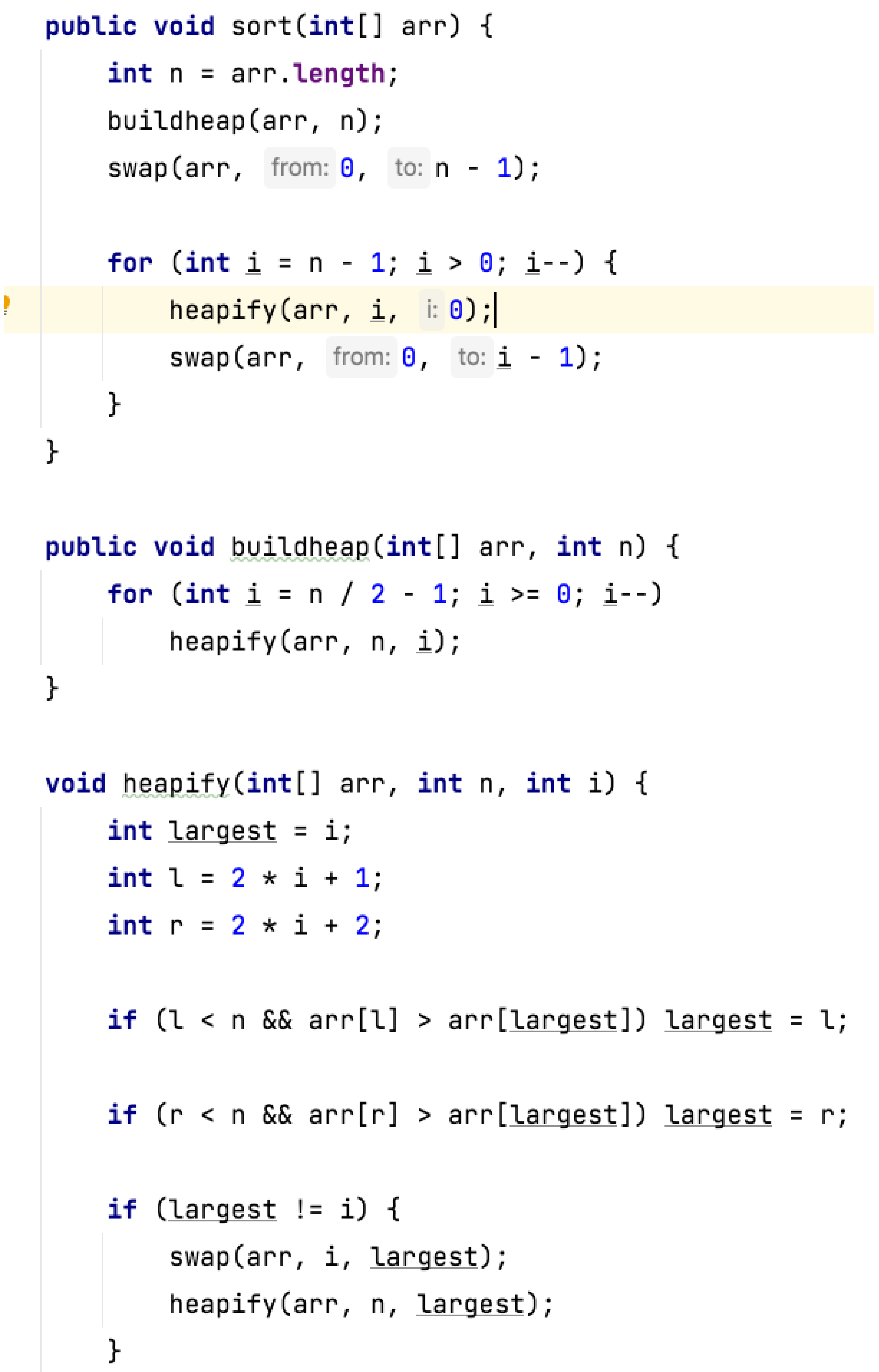


```java
private static void qSort(int[] arr) {
    int r = new Random().nextInt(arr.length);
    swap(arr, from: 0, r);
    qSort(arr, l: 0, h: arr.length - 1);
}

private static void qSort(int[] arr, int l, int h) {
    if (l < h) {
        int p = hoarePartition(arr, l, h);
        qSort(arr, l, p); // This is important  (p - 1) is bad option, that will exclude elements.
        qSort(arr, l: p + 1, h);
    }
}
```

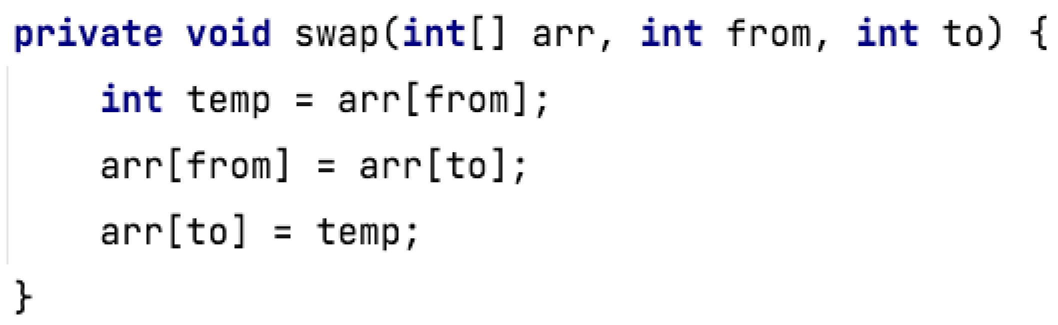


```java
private static int hoarePartition(int[] arr, int l, int h) {
    int pivot = arr[l];
    int i = l - 1; // since do while go one step back
    int j = h + 1; // since do while go one step ahead
    while (true) {
        do { // do while important, else infinite loop happens after
            // first swap, need to increment and check condition
            i++;
        } while (arr[i] < pivot); // <= is a bad option,
        // as the first element will get stuck in its position
        // and sorting will be wrong
        do {
            j--;
        } while (arr[j] > pivot);
        if (i >= j) return j; // no need to do for == as it is same value/index
         // return "j" is key, returning "i" will include higher value to left side
        // and right side of pivot is not all sorted yet, so final result won't be sorted
        // safer bet is to return "j"
        swap(arr, i, j);
    }
}
```

## HeapSort (N * log(N)) for all cases

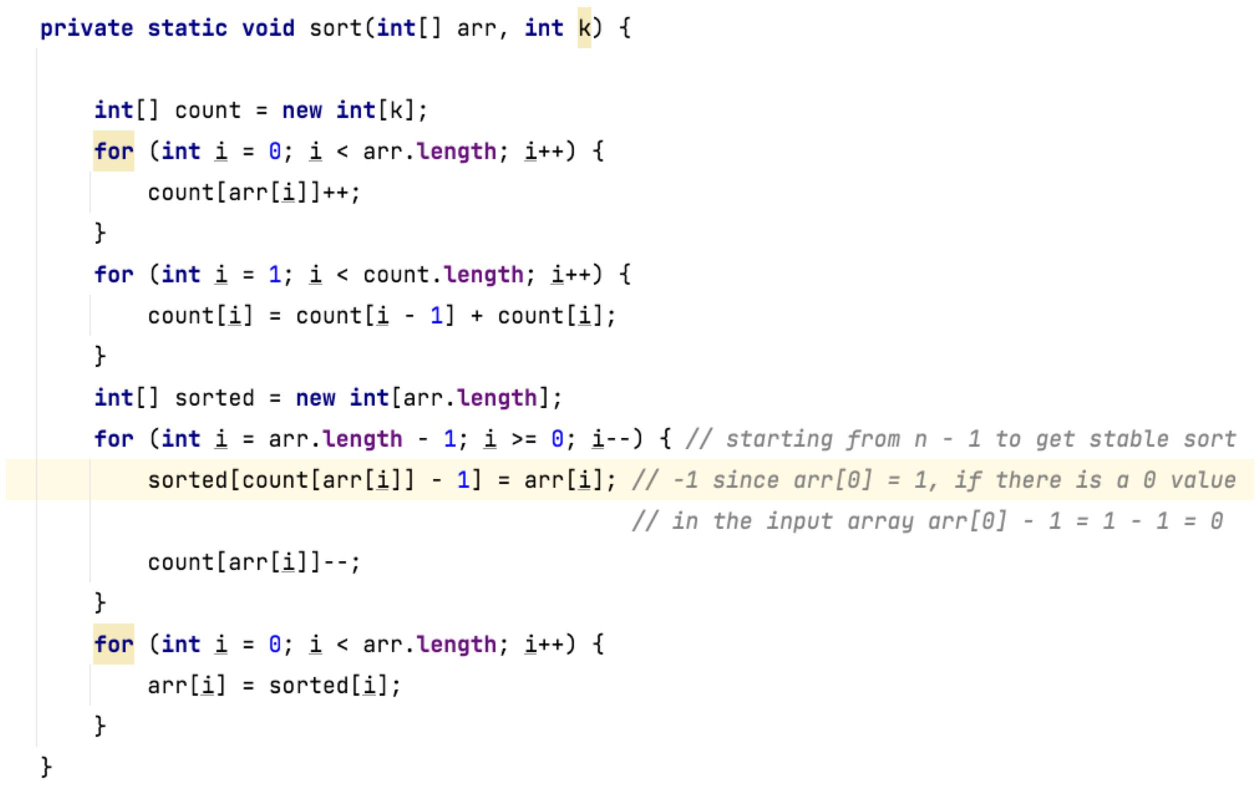


```
public void sort(int[] arr) {
    int n = arr.length;
    buildheap(arr, n);
    swap(arr, from: 0, to: n - 1);

    for (int i = n - 1; i > 0; i--) {
        heapify(arr, i, i: 0);
        swap(arr, from: 0, to: i - 1);
    }
}

public void buildheap(int[] arr, int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
}

void heapify(int[] arr, int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < n && arr[l] > arr[largest]) largest = l;

    if (r < n && arr[r] > arr[largest]) largest = r;

    if (largest != i) {
        swap(arr, i, largest);
        heapify(arr, n, largest);
    }
```



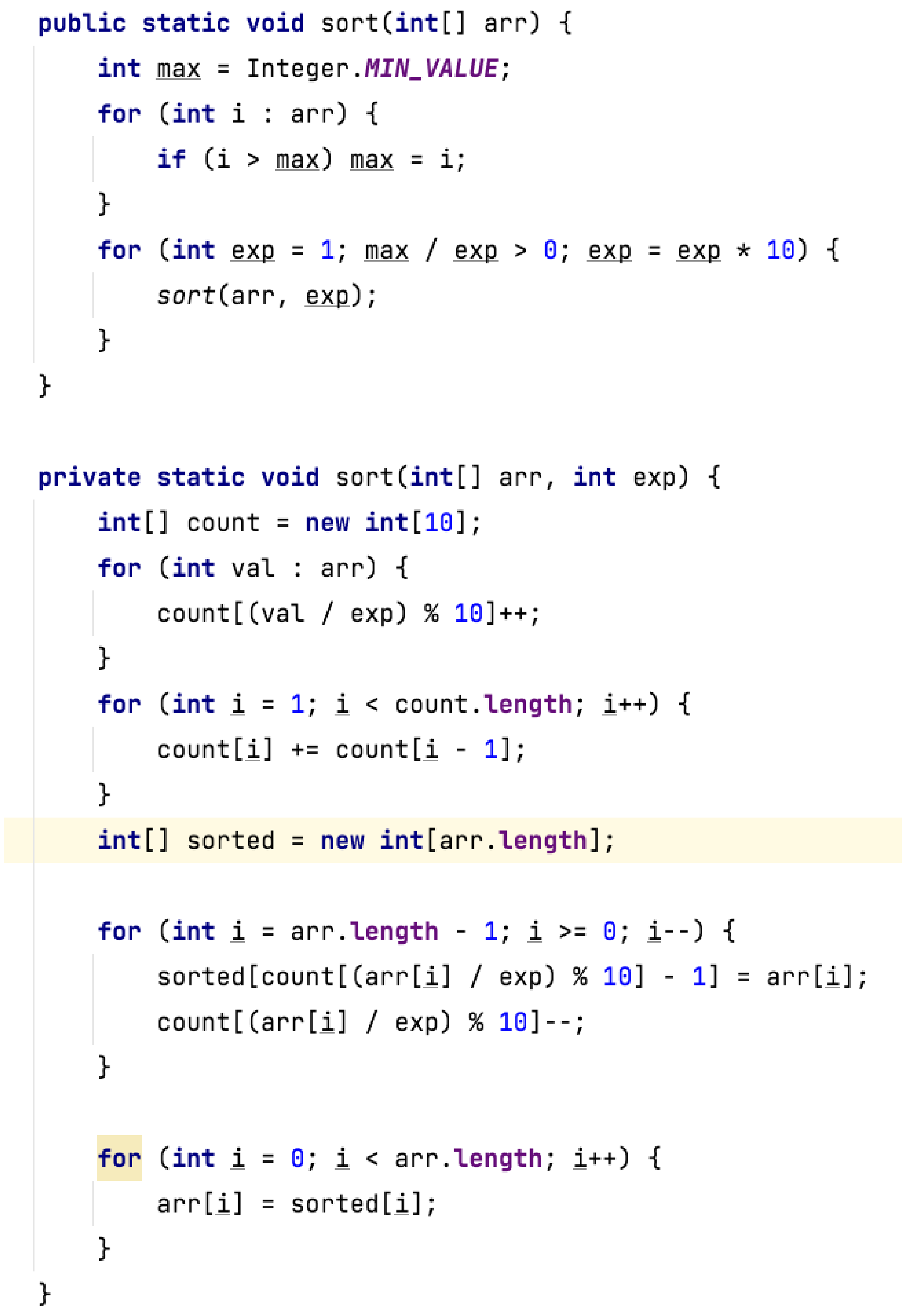
```java
private void swap(int[] arr, int from, int to) {
    int temp = arr[from];
    arr[from] = arr[to];
    arr[to] = temp;
}
```

**Counting Sort O(N+K) space o(n+k)**



```java
private static void sort(int[] arr, int k) {

    int[] count = new int[k];
    for (int i = 0; i < arr.length; i++) {
        count[arr[i]]++;
    }
    for (int i = 1; i < count.length; i++) {
        count[i] = count[i - 1] + count[i];
    }
    int[] sorted = new int[arr.length];
    for (int i = arr.length - 1; i >= 0; i--) { // starting from n - 1 to get stable sort
        sorted[count[arr[i]] - 1] = arr[i]; // -1 since arr[0] = 1, if there is a 0 value
                                            // in the input array arr[0] - 1 = 1 - 1 = 0
        count[arr[i]]--;
    }
    for (int i = 0; i < arr.length; i++) {
        arr[i] = sorted[i];
    }
}
```

**Radix Sort time complexity = O ( d * (n+10) ) , where d = number of digits in largest number and space complexity is O(n)**



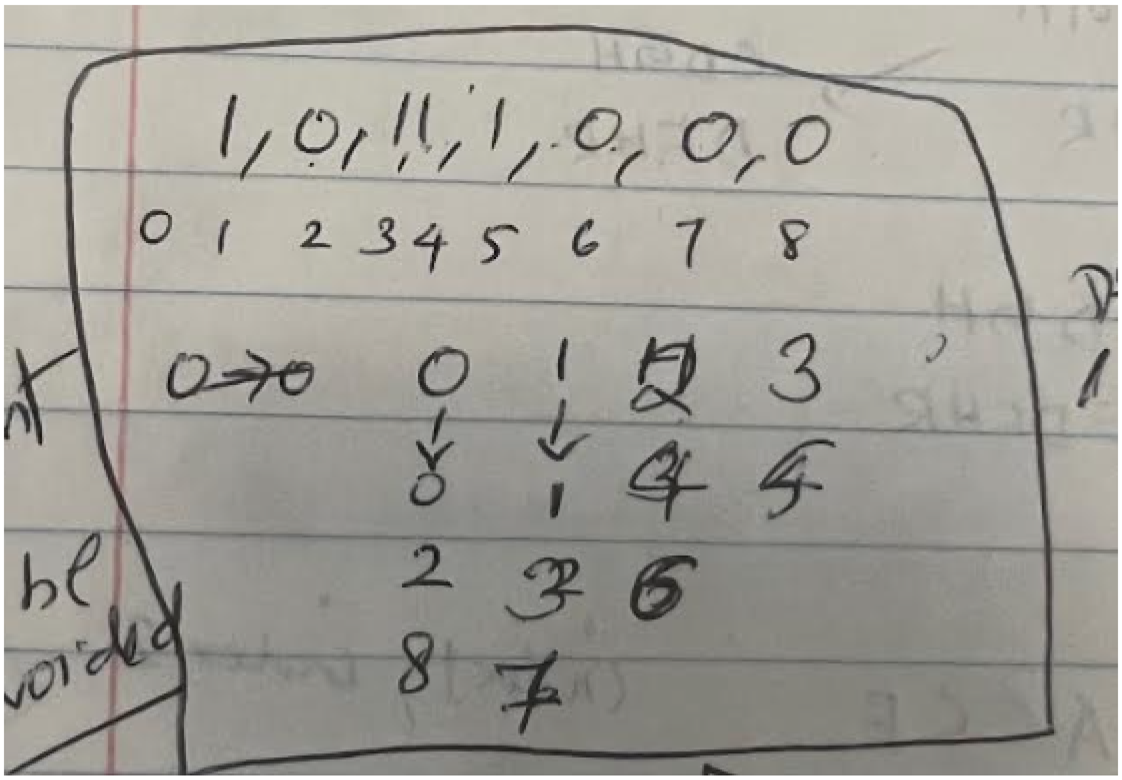
```java
public static void sort(int[] arr) {
    int max = Integer.MIN_VALUE;
    for (int i : arr) {
        if (i > max) max = i;
    }
    for (int exp = 1; max / exp > 0; exp = exp * 10) {
        sort(arr, exp);
    }
}

private static void sort(int[] arr, int exp) {
    int[] count = new int[10];
    for (int val : arr) {
        count[(val / exp) % 10]++;
    }
    for (int i = 1; i < count.length; i++) {
        count[i] += count[i - 1];
    }
    int[] sorted = new int[arr.length];

    for (int i = arr.length - 1; i >= 0; i--) {
        sorted[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    for (int i = 0; i < arr.length; i++) {
        arr[i] = sorted[i];
    }
}
```

**Bucket sort  - When data is uniformly distributed from ranges e.g. 1-100, 200-300 ranges, etc. separate by bucket and sort each bucket. Space = n, time = n log n**

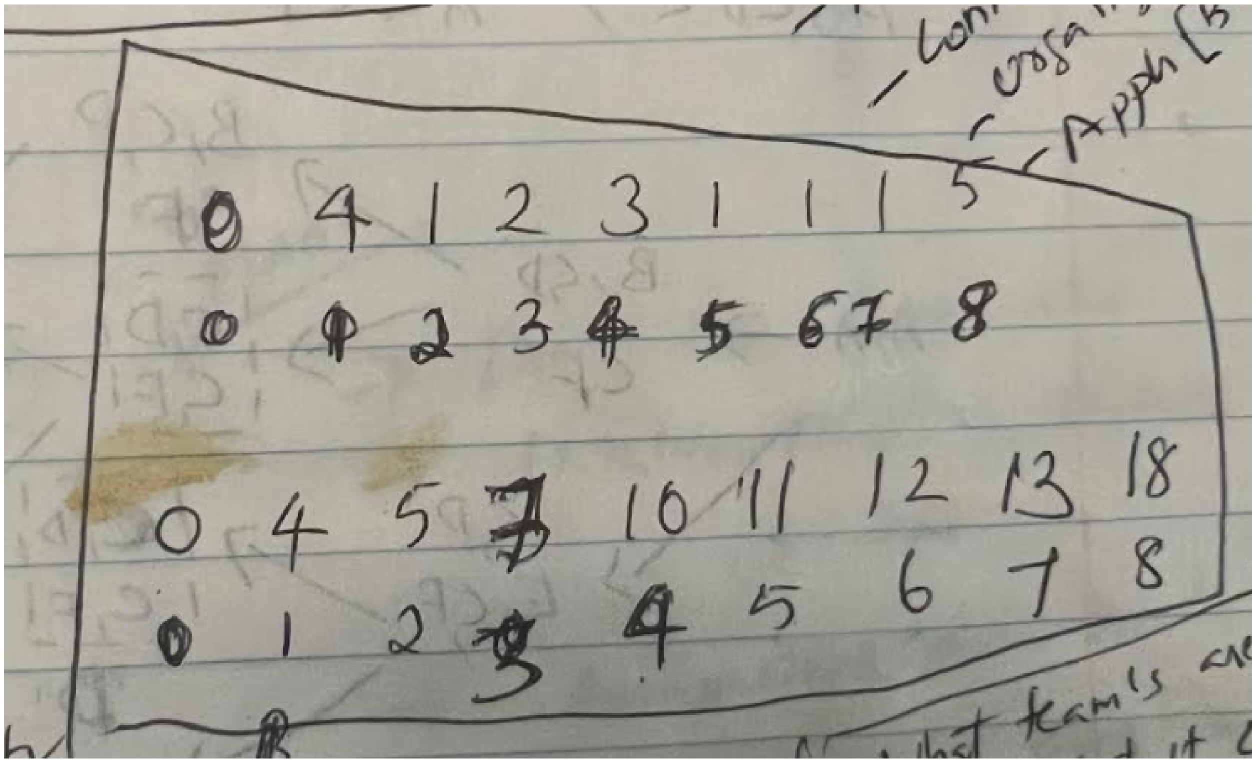
**When memory writes are costly like EPOM, the best algorithm is selection sort, though it has n * 2 CPU memory operations or swaps are N only**

**For a small array of 20 elements are small, insertion sort works better even though it is N * 2**

**Hashing - 0s and 1s**



**Hashing - sum**



**Leet code documentation:**

1. ReverseLinkedList

To reverse a linked list with two pointers

2. ComparatorTest

All sort of functional comparator

3. Container with most water Link

Two pointers from both side and move left and right

4. SplitArrayLargestSum - Hard problem and binary search did well here
5. SwappingNodeInLinkedList Link

The solution was to swap the value not to adjust the pointers