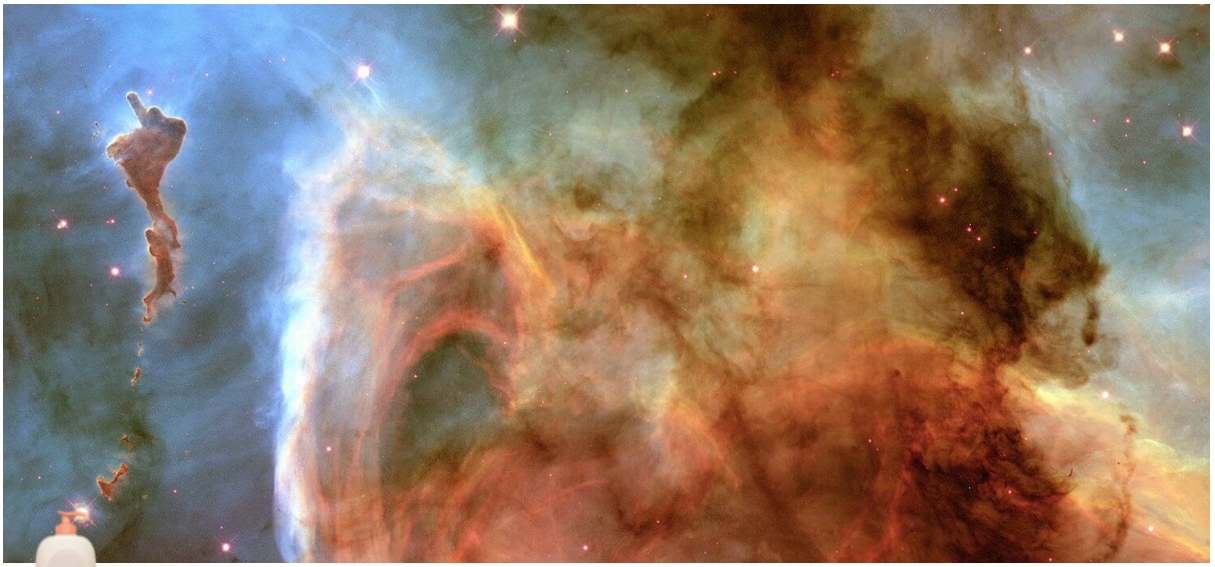

Solutions

Data Structures & Algorithms Labs
Week 3: Sorting

These exercises will allow you to test your knowledge of the lecture material from week 2 (sorting).

1 Programming Exercises

Once you have solved each of these exercises, try to figure out the (average/worst case) time and space complexity of your solution, and whether more efficient solutions exist.

► **1.1.** Write a function that takes two sorted arrays, both in non-decreasing order, and outputs a single sorted array containing all the elements of both input arrays. Can you do it in linear time?

Example: $[1, 3, 4, 5, 6, 8, 9], [1, 2, 3, 4] \to [1, 1, 2, 3, 3, 4, 4, 5, 6, 8, 9]$ ◄

Solution. The idea behind this solution is that we loop through the both arrays (`arr1` and `arr2`) at once, keeping track of both their indices (`i` and `j`), and we use `i+j` to index the merged array. Each loop iteration we compare `arr1[i]` to `arr2[j]`, and set `merged[i+j]` to whichever is smaller (`arr1[i]` if equal). If `arr1[i]` was added, then we increment `i`, and if `arr2[j]` was added we increment `j`. We therefore always increment either `i` or `j`, but never both, such that the merged index `i+j` is guaranteed to increase by one every iteration. We continue this until we reach the end of the smaller array, and then we leftover elements of the other array in order. Finally, since both arrays are sorted we know that each newly added element is no smaller than the previously added element, and so the merged array will also be sorted.

```
int[] mergeArrays(int[] arr1, int[] arr2) {
    int n = arr1.length, m = arr2.length;
    int[] merged = new int[n + m];
    int i = 0, j = 0;

    // Merge arrays across their overlapping region
    while (i < n && j < m) {
        if (arr1[i] <= arr2[j]) {
            merged[i+j] = arr1[i];
            i++;
        } else{
            merged[i+j] = arr2[j];
            j++;
        }
    }

    // Append remaining elements of whichever is longer
    while (i < n) {
        merged[i+j] = arr1[i];
        i++;
    }
    while (j < m) {
        merged[i+j] = arr2[j];
        j++;
    }

    return merged;
}
```

You can also write this function in the following compact (but perhaps less readable) form:

```
int[] mergeArrays(int[] arr1, int[] arr2) {
    int n = arr1.length, m = arr2.length;
    int[] merged = new int[n + m];
    int i = 0, j = 0;

    // Merge arrays across their overlapping region
    while (i < n && j < m) {
        merged[i+j] = (arr1[i] <= arr2[j]) ? arr1[i++] : arr2[j++];
    }

    // Append remaining elements of whichever is longer
    while (i < n) merged[i+j] = arr1[i++];
    while (j < m) merged[i+j] = arr2[j++];

    return merged;
}
```

Time complexity: $O(m+n)$. Space complexity: $O(m+n)$. □

► **1.2.** Write a stable implementation of the quicksort algorithm. Stable means that all pairs of objects with equal sorting keys will appear in the same relative order in the output as they do in the input. In other words, if two elements are equivalent then they won't be swapped with each other. For this exercise you *do not* need to use an in-place partitioning function! ◄

Solution. To make the quickSort algorithm stable, we need to focus on the partition algorithm. The function that handles the recursion can remain as before:

```
void quickSortBetween(double[] a, int left, int right) {
    if (left < right) {
        int pivotIndex = partition(a, left, right);
        quickSortBetween(a, left, pivotIndex - 1);
        quickSortBetween(a, pivotIndex + 1, right);
    }
}

void quickSort(double[] a) {
    quickSortBetween(a, 0, a.length-1);
}
```

We will use a partition function that does not reorder the array in-place. Instead we will allocate two temporary arrays: `lower` to store elements no greater than the pivot, and `higher` to store elements no less than the pivot. Elements equal to the pivot will be placed in `lower` if their index is less than that of the pivot, and `higher` if their index is greater. We do this by looping through the input array and using two extra indices to keep track of position in the temporary arrays (e.g. whenever we add an element to `lower`, we increment its index). At the end of the loop we then use the final value of these indices to loop over the temporary arrays and put the elements back in the input array in their partitioned order.

```
// 1. Picks a pivot between left and right
// 2. Orders elements between left and right into
// three sections: <= pivot, pivot, >= pivot
// 3. Returns new pivot index
int partition(double[] a, int left, int right) {

    // Incorrectly called if left < right
    if (left >= right) {
        return -1;
    }

    int n = a.length;
    double[] lower = new double[n-1];
    double[] higher = new double[n-1];
```

```
    // Pick middle element as pivot (arbitrary)
    int pivotIndex = left + (right - left)/2;
    double pivot = a[pivotIndex];

    int j = 0, k = 0;
    for (int i = left; i <= right; i++) {

        // Cast difference to an integer, allows us to check stability
        int difference = (int)(a[i] - pivot);

        if (difference < 0) {
            lower[j] = a[i];
            j++;
        }
        if (difference > 0) {
            higher[k] = a[i];
            k++;
        }

        // If elements are equal sort by index, and skip over pivot
        if (difference == 0 && i < pivotIndex) {
            lower[j] = a[i];
            j++;
        }
        if (difference == 0 && i > pivotIndex) {
            higher[k] = a[i];
            k++;
        }
    }

    for (int i = 0; i < j; i++) {
        a[left + i] = lower[i];
    }

    a[left + j] = pivot;

    for (int i = 0; i < k; i++) {
        a[left + i + j + 1] = higher[i];
    }

    return left + j;
}
```

Note that we are sorting the array of doubles using integer comparison. This is so that we can check stability: if we have two elements `1.0` and `1.1` in the array in that order, then they should stay in that order after sorting (this also works for negative numbers).

Average time complexity: $O(\mathtt{n}\log\mathtt{n})$. Average space complexity: $O(\mathtt{n})$ □

► **1.3.** Write a function that takes an array and returns an array with only the unique elements from the input array (duplicates removed).

Example: $[1,9,3,1,1,5,6,9,5] \to [1,3,5,6,9]$ ◄

Solution. Let the input array be `arr` and its length be `n`. The key to this questions is to first sort `arr` using our stable quicksort function (or any other sorting algorithm). Once this is done duplicate entries will be grouped together. We then create a temporary array `temp` of length `n` which will store the unique entries, and we set its first element to the first element `arr`. Next, we create two integers `i` and `j` to index `temp` and `arr`, respectively, and we loop through `arr` using `j`. Whenever we encounter an input element `arr[j]` that is different to current element of the temporary array, `temp[i]`, we increment the temporary array index (`i++`) and then set `temp[i]`

to the new unique element we have found. We know that this is a new unique element because the input array has been sorted. Once this loop is finished `temp` will have its first `i+1` elements initialized to the unique elements of `arr`, and its remaining `n - i - 1` elements will be uninitialized. If we were using dynamic arrays (ArrayList in Java), we could then just resize `temp` to be of size `i+1`, and would have $O(1)$ space complexity. Since we are using static arrays, we need to create a new array of length `i+1`, and set these elements equal to the initialized elements of `temp`.

```
int[] removeDuplicates(int[] arr) {
    int n = arr.length;
    if (n == 0) return arr;
    stableQuickSort(arr);

    int[] temp = new int[n];
    temp[0] = arr[0];
    int i = 0;
    for (int j = 1; j < n; j++) {
        if (arr[j] != temp[i]) {
            i++;
            temp[i] = arr[j];
        }
    }

    int[] newArr = new int[i+1];
    for (int j = 0; j < i+1; j++) {
        newArr[j] = temp[j];
    }

    return newArr;
}
```

Average time complexity: $O(n \log n)$ (dominated by initial sorting step). Space complexity: $O(n)$. □

► **1.4.** Write a function that takes two input arrays, `arr1` and `arr2`, of lengths `m` and `n` respectively, and outputs `true` if every element of `arr1` is in `arr2`, and `false` otherwise.

Example 1: $[2, 5, 3], [3, 2, 1, 4, 5] \rightarrow$ `true`

Example 2: $[1, 6], [3, 2, 1, 4, 5] \rightarrow$ `false` ◄

Solution. Our first step is to sort `arr2` ($O(n \log n)$), so that we can search it. We then loop through `arr1` and binary search `arr2` for the current element of `arr1` each iteration. If any of searches returns `false`, then our function returns `false`. Then if we get to the end of the loop we know that all the searches returned `true` and so our function returns `true`.

```
// Leaving arr1 unsorted
boolean isSubset(int[] arr1, int[] arr2) {
    int m = arr1.length;
    quickSort(arr2);
    for (int i = 0; i < m; i++) {
        found = binarySearch(arr2, arr1[i])
        if (!found) return false;
    }
    return true;
}
```

Time complexity: $O((m + n) \log n)$. Space complexity: $O(1)$.

We can optionally also sort `arr1` (adding $O(m \log m)$ time) in order to skip over any duplicates in `arr1`. In general this will not actually change the asymptotic complexity in the search step ($O(m \log n)$), since the number of unique elements in `arr1` will generally scale linearly with its length (`m`).

```
// Sorting arr1 and skipping duplicates - adds O(mlogm)
boolean isSubset(int[] arr1, int[] arr2) {
```

```
    int m = arr1.length, n = arr2.length;
    if (n == 0) return arr1.length == 0;

    quickSort(arr1);
    quickSort(arr2);

    boolean found = binarySearch(arr2, arr1[0]);
    if (!found) return false;
    for (int i = 1; i < m; i++) {
        if (arr1[i] != arr1[i-1]) {
            found = binarySearch(arr2, arr1[i]);
            if (!found) return false;
        }
    }

    return true;
}
```

Time complexity: $O(m \log m + (m+n) \log n)$. Space complexity: $O(\log m + \log n)$.

Note that this question and the previous one can be solved more simply and efficiently with the use of HashSets, which we haven't covered yet in this course. □

► **1.5.** Write a function that takes an `m*n` matrix of integers from `0-9`, and returns the same matrix with its rows sorted lexicographically in increasing order, as if each row were a single number.

Example: $\begin{bmatrix} 1 & 7 & 8 & 2 \\ 1 & 3 & 3 & 3 \\ 9 & 1 & 2 & 3 \\ 5 & 6 & 2 & 9 \\ 1 & 7 & 1 & 9 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 3 & 3 & 3 \\ 1 & 7 & 1 & 9 \\ 1 & 7 & 8 & 2 \\ 5 & 6 & 2 & 9 \\ 9 & 1 & 2 & 3 \end{bmatrix}$

◄

Solution. The solution below uses a concatenate function to convert an array to a single number and store it in `array[0]`. We then have a deconcatenate function which takes `array[0]` and reconstructs its original value by dropping digits one-by-one from last element in `array` backwards. This could allow us to achieve $O(1)$ space complexity, as opposed to $O(m)$ if we had stored these numbers in a new array.

```
public class DigitArray {

    static void concatenate(int[] array) throws Exception {
        int n = array.length;
        if (array[0] < 0) {
            throw new Exception("Can't concatenate negative digits");
        }
        if (array[0] > 9) {
            throw new Exception("Can't concatenate multi-digit numbers");
        }
        for (int i = 1; i < n; i++) {
            if (array[i] < 0) {
                throw new Exception("Can't concatenate negative digits");
            }
            if (array[i] > 9) {
                throw new Exception("Can't concatenate multi-digit numbers");
            }
            array[0] = 10 * array[0] + array[i];
        }
    }

    static void deconcatenate(int[] array) throws Exception {
        int n = array.length;
        for (int i = n-1; i > 0; i--) {
```

```
            int diff = array[0] - array[i];
            if (diff % 10 != 0) {
                throw new Exception("array[0] does not match concatenation of remaining digits");
            }
            array[0] = (array[0] - array[i])/10;
        }
    }

}
```

Once we have these functions, we then just have to take the input matrix, concatenate each of its rows, sort these rows according to their first elements, and then deconcatenate the rows at the end. We could write this sorting function by hand (to achieve constant space complexity for example), but for simplicity we are using an inbuilt Java sorting function, Arrays.sort, with a custom comparator function to sort by first element of each row. This uses a version of merge sort with average time complexity $O(\mathtt{m}\log\mathtt{m})$ and space complexity $O(\mathtt{m})$.

```
void lexicographicDigitMatrixSort(int[][] matrix) {
    int m = matrix.length;

    for (int i = 0; i < m; i++) {
        try {
            DigitArray.concatenate(matrix[i]);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

    Arrays.sort(matrix, Comparator.comparingInt(row -> row[0]));

    for (int i = 0; i < m; i++) {
        try {
            DigitArray.deconcatenate(matrix[i]);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Time complexity: $O(\mathtt{m}*\mathtt{n}+\mathtt{m}\log\mathtt{m})$. Space complexity: $O(\mathtt{m})$.

Another (less efficient) solution is to sort the rows by first elements, breaking ties with the second elements, breaking those ties with the third elements, and so on until the last elements. Again, for simplicity we implement this using Arrays.sort and a custom comparator.

```
void LexicographicMatrixSort(int[][] matrix) {
    int n = matrix.length;
    int m = matrix[0].length;

    // Sort rows lexicographically

    // A lambda expression used to compare arrays
    // since a matrix is an array of arrays
    Arrays.sort(matrix, (a,b) -> {
        // a and b are array
        for (int i = 0; i<m; i++) {
            if (a[i] != b[i]) {
                return Integer.compare(a[i], b[i]);
            }
        }
        // return 0 to ensure equality
        return 0;
    });
}
```

Time complexity: $O(\texttt{m*n}\log(\texttt{m*n}))$. Space complexity: $O(\texttt{m})$. □

► **1.6.** Write a function that takes an array of `n` time intervals, `(a,b)`, where `0<a<b`, and sorts them by:

- The earliest start.
- If tied, by shortest duration.

Example: $[(5,7),(5,6),(3,4),(2,8),(3,6),(4,5)] \to [(2,8),(3,4),(3,6),(4,5),(5,6),(5,7)]$ ◄

Solution. This is actually quite similar to the previous exercise, since we can think of an array of intervals as a matrix with only two columns (and the added restriction that the first element of each row is less than the second). It appears very different at first glance since we are sorting by shortest duration `a-b` in the case of ties rather than by the second element `b`. However, if the start times are equal then sorting by shortest duration is exactly the same as sorting by end time in ascending order! We will make use of this in the solution. Unlike the previous question, we no longer have the restriction of single digit numbers here, so our concatenation solution wouldn't work. Another difference from the previous question is that this question would still make sense if we replace these with floats or doubles. Nevertheless, we will assume integers for simplicity. We could again use in build sorting with a custom comparator, but instead let's see how we would do it from scratch. We start by defining a custom class to store an interval (not strictly necessary, but nice to have).

```
public class Interval {
    private int start, end;

    Interval(int start, int end)  {
        try {
            if (start > end) {
                throw new Exception("End of interval must not be before start");
            } else {
                this.start = start;
                this.end = end;
            }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

    Interval(Interval i) {
        this.start = i.start;
        this.end = i.end;
    }

    int start() {
        return this.start;
    }

    int end() {
        return this.end;
    }

    void copy(Interval i) {
        start = i.start;
        end = i.end;
    }

    @Override
    public String toString() {
        return "(" + start + ", " + end + ")";
    }
}
```

Now, we will adapt our stable quickSort algorithm to this new type. The recursion handler is essentially the same.

```
void quickSortIntervalsBetween(Interval[] a, int left, int right) {
    if(left < right) {
        int pivotIndex = partitionIntervals(a, left, right);
        quickSortIntervalsBetween(a, left, pivotIndex - 1);
        quickSortIntervalsBetween(a, pivotIndex + 1, right);
    }
}

void quickSortIntervals(Interval[] a) {
    quickSortIntervalsBetween(a, 0, a.length-1);
}
```

The part we have to change is the partition algorithm. Before we compared the indices in the case of ties in order to ensure stability. Now we will first compare end times, and then only when these are equal will we compare indices. Note that we have used `new Interval` to copy intervals into the temporary arrays. This is to ensure that we have a *deep copy* where new class instances are created with copies of the underlying data, rather than a *shallow copy* where only the memory addresses of the original class instances are copied and stored in the temporary arrays.

```
int partitionIntervals(Interval[] a, int left, int right) {

    // Only correct if left < right
    if (left >= right) {
        return -1;
    }

    int n = a.length;
    Interval[] lower = new Interval[n-1];
    Interval[] higher = new Interval[n-1];

    // Pick middle element as pivot (arbitrary)
    int pivotIndex = left + (right - left)/2;
    Interval pivot = new Interval(a[pivotIndex]);
    int pivotStart = a[pivotIndex].start();
    int pivotEnd = a[pivotIndex].end();

    int j = 0, k = 0;
    for(int i = left; i <= right; i++) {

        int start = a[i].start();
        if(start < pivotStart) {
            lower[j] = new Interval(a[i]);
            j++;
        }
        if(start > pivotStart) {
            higher[k] = new Interval(a[i]);
            k++;
        }

        // If starts are equal sort by end
        // Same as sorting by duration when starts are equal
        if (start == pivotStart) {
            int end = a[i].end();
            if (end < pivotEnd) {
                lower[j] = new Interval(a[i]);
                j++;
            }
            if (end > pivotEnd) {
                higher[k] = new Interval(a[i]);
                k++;
            }
            // If both are equal then sort by index and skip pivot
```

```
            if (end == pivotEnd && i < pivotIndex) {
                lower[j] = new Interval(a[i]);
                j++;
            }
            if (end == pivotEnd && i > pivotIndex) {
                higher[j] = new Interval(a[i]);
                k++;
            }
        }
    }

    for(int i = 0; i < j; i++) {
        a[left + i].copy(lower[i]);
    }

    a[left + j].copy(pivot);

    for(int i = 0; i < k; i++) {
        a[left + i + j + 1].copy(higher[i]);
    }

    return left + j;
}
```

Time complexity: $O(\mathtt{n} \log \mathtt{n})$. Space complexity: $O(\mathtt{n})$. □

2 Theory Exercises

► **2.1.** Compute the sum $\sum_{j=k+m}^{n-p+2} 1$, where k, m, n, p are constants. ◄

Solution. The sum equals

$$\sum_{j=k+m}^{n-p+2} 1 = n - p - k - m + 3. \qquad \square$$

► **2.2.** Compute the double sum $\sum_{i=k}^{n-3} \sum_{j=m}^{n^2} 1$, where k, m, n are constants. ◄

Solution. The sum equals

$$\sum_{i=k}^{n-3} \sum_{j=m}^{n^2} 1 = \sum_{i=k}^{n-3} (n^2 - m + 1) = (n - k - 2)(n^2 - m + 1). \qquad \square$$

► **2.3.** In each of the following cases, write the given expression compactly using summation notation, $\sum$ (there is more than one correct answer for all cases).

1. $A = 4 + 9 + 16 + 25 + 36$;
2. $B = 2^2 + 3^4 + 4^6 + 5^8 + 6^{10}$;
3. $C = 2 - 3 + 4 - 5 + 6 - 7 + 8 - 9$;
4. $D = -4 + 6 - 8 + 10 - 12 + 14$. ◄

Solution. We give one solution for each case.

1. $A = \sum_{i=2}^{6} i^2$;
2. $B = \sum_{i=1}^{5} (i+1)^{2i}$;
3. $C = \sum_{i=2}^{9} i \cdot (-1)^i$;
4. $D = \sum_{i=2}^{7} 2i \cdot (-1)^{i+1}$. $\square$

► **2.4.** Show that the complexity of sorting an array of `n` elements with insertion sort is $O(\mathtt{n}^2)$. To show the upper bound of $O(\mathtt{n}^2)$ consider an inversely sorted list. ◄

Solution. In the worst-case scenario, for the i-th element ($i = 1, 2, \ldots, n$), we need to make $i - 1$ comparisons. This happens when the element to be inserted is smaller than all of the $i - 1$ elements in the sorted part. The total number of steps in the worst-case is therefore

$$\sum_{i=1}^{n} (i - 1) = 0 + 1 + 2 + \ldots + (n - 1) = \frac{n^2 - n}{2} \in O(n^2). \qquad \square$$

► **2.5.** Perform the final step of mergesort on the following sorted arrays:

$$2, 5, 8, 10 \text{ and } 7, 9, 11, 13, 15.$$ ◄

Solution. We repeatedly compare the first elements of the two arrays and put the smallest into the merged array. When one array empties, we append the other at the end of the merged array. The successive steps are the following:

Compare	Add
2 and 7	2
5 and 7	5
8 and 7	7
8 and 9	8
10 and 9	9
10 and 11	10

Since the first array has emptied, we append the merged array with $11, 13, 15$. The merged array is

$$2, 5, 7, 8, 9, 10, 11, 13, 15.$$ □

► **2.6.** Consider the following queue of playing cards.

$\longrightarrow$ | 3♠ | A♡ | 2♣ | 3♣ | 2♢ | A♠ | A♢ | 3♡ | 3♢ | A♣ | 2♡ | 2♠ | $\longrightarrow$

Use bin sorting to sort this queue, so that the final queue of playing cards will be ordered first by suit (order is ♡, ♢, ♠, ♣), and then by face-value (order is $A, 2, 3$). ◄

Solution. First, we dequeue each element, and place it into a bin according to its face value. Since we have 3 face values and 4 suits, we need 3 bins with 4 elements each.

Bin of A: $\longrightarrow$ | A♡ | A♠ | A♢ | A♣ | $\longrightarrow$

Bin of 2: $\longrightarrow$ | 2♣ | 2♢ | 2♡ | 2♠ | $\longrightarrow$

Bin of 3: $\longrightarrow$ | 3♠ | 3♣ | 3♡ | 3♢ | $\longrightarrow$

We then use dequeue to concatenate these three bins into one long queue with the following order: first the bin of A, then the bin of 2, then the bin of 3.

$\longrightarrow$ | 3♠ | 3♣ | 3♡ | 3♢ | 2♣ | 2♢ | 2♡ | 2♠ | A♡ | A♠ | A♢ | A♣ | $\longrightarrow$

We finally perform another bin sorting but this time, we place the elements according to their suits. Since we have 4 suits and 3 face values, we need 4 bins with 3 elements each.

Bin of ♡: $\longrightarrow$ | 3♡ | 2♡ | A♡ | $\longrightarrow$

Bin of ♢: $\longrightarrow$ | 3♢ | 2♢ | A♢ | $\longrightarrow$

Bin of ♠: $\longrightarrow$ | 3♠ | 2♠ | A♠ | $\longrightarrow$

Bin of ♣: $\longrightarrow$ | 3♣ | 2♣ | A♣ | $\longrightarrow$

Finally, we use dequeue to concatenate these bins into one long queue in the following order: first the bin of ♡, then the bin of ♢, then the bin of ♠, then the bin of ♣.

$\longrightarrow$ | 3♣ | 2♣ | A♣ | 3♠ | 2♠ | A♠ | 3♢ | 2♢ | A♢ | 3♡ | 2♡ | A♡ | $\longrightarrow$

The sorting we want is achieved. □

► **2.7.** (Non-examinable) Pigeonhole sort in-place the array $2, 4, 0, 1, 3$. ◄

Solution. For each index $(0, 1, 2, 3, 4)$, we make a swap so the element in the index position goes to its correct position. The successive steps are the following:

0	1	2	3	4
$\underline{2}$	4	$\underline{0}$	1	3
0	$\underline{4}$	2	1	$\underline{3}$
0	$\underline{3}$	2	$\underline{1}$	4
0	1	2	3	4

In the final array, each element equals its index, so we are done. □