10

Sorting and Searching

Understanding the common sorting and searching algorithms is incredibly valuable, as many sorting and searching problems are tweaks of the well-known algorithms. A good approach is therefore to run through the different sorting algorithms and see if one applies particularly well.

For example, suppose you are asked the following question: Given a very large array of `Person` objects, sort the people in increasing order of age.

We're given two interesting bits of knowledge here:

1. It's a large array, so efficiency is very important.
2. We are sorting based on ages, so we know the values are in a small range.

By scanning through the various sorting algorithms, we might notice that bucket sort (or radix sort) would be a perfect candidate for this algorithm. In fact, we can make the buckets small (just 1 year each) and get $O(n)$ running time.

▸ Common Sorting Algorithms

Learning (or re-learning) the common sorting algorithms is a great way to boost your performance. Of the five algorithms explained below, Merge Sort, Quick Sort and Bucket Sort are the most commonly used in interviews.

Bubble Sort | Runtime: $O(n^2)$ average and worst case. Memory: $O(1)$.

In bubble sort, we start at the beginning of the array and swap the first two elements if the first is greater than the second. Then, we go to the next pair, and so on, continuously making sweeps of the array until it is sorted. In doing so, the smaller items slowly "bubble" up to the beginning of the list.

Selection Sort | Runtime: $O(n^2)$ average and worst case. Memory: $O(1)$.

Selection sort is the child's algorithm: simple, but inefficient. Find the smallest element using a linear scan and move it to the front (swapping it with the front element). Then, find the second smallest and move it, again doing a linear scan. Continue doing this until all the elements are in place.

Merge Sort | Runtime: $O(n \log(n))$ average and worst case. Memory: Depends.

Merge sort divides the array in half, sorts each of those halves, and then merges them back together. Each of those halves has the same sorting algorithm applied to it. Eventually, you are merging just two single-element arrays. It is the "merge" part that does all the heavy lifting.

The merge method operates by copying all the elements from the target array segment into a helper array, keeping track of where the start of the left and right halves should be (`helperLeft` and `helperRight`). We then iterate through `helper`, copying the smaller element from each half into the array. At the end, we copy any remaining elements into the target array.

```
1  void mergesort(int[] array) {
2    int[] helper = new int[array.length];
3    mergesort(array, helper, 0, array.length - 1);
4  }
5
6  void mergesort(int[] array, int[] helper, int low, int high) {
7    if (low < high) {
8      int middle = (low + high) / 2;
9      mergesort(array, helper, low, middle); // Sort left half
10     mergesort(array, helper, middle+1, high); // Sort right half
11     merge(array, helper, low, middle, high); // Merge them
12   }
13 }
14
15 void merge(int[] array, int[] helper, int low, int middle, int high) {
16   /* Copy both halves into a helper array */
17   for (int i = low; i <= high; i++) {
18     helper[i] = array[i];
19   }
20
21   int helperLeft = low;
22   int helperRight = middle + 1;
23   int current = low;
24
25   /* Iterate through helper array. Compare the left and right half, copying back
26    * the smaller element from the two halves into the original array. */
27   while (helperLeft <= middle && helperRight <= high) {
28     if (helper[helperLeft] <= helper[helperRight]) {
29       array[current] = helper[helperLeft];
30       helperLeft++;
31     } else { // If right element is smaller than left element
32       array[current] = helper[helperRight];
33       helperRight++;
34     }
35     current++;
36   }
37
38   /* Copy the rest of the left side of the array into the target array */
39   int remaining = middle - helperLeft;
40   for (int i = 0; i <= remaining; i++) {
41     array[current + i] = helper[helperLeft + i];
42   }
43 }
```

You may notice that only the remaining elements from the left half of the helper array are copied into the target array. Why not the right half? The right half doesn't need to be copied because it's *already* there.

Consider, for example, an array like `[1, 4, 5 || 2, 8, 9]` (the "`||`" indicates the partition point). Prior to merging the two halves, both the helper array and the target array segment will end with `[8, 9]`. Once we copy over four elements (`1`, `4`, `5`, and 2) into the target array, the `[8, 9]` will still be in place in both arrays. There's no need to copy them over.

The space complexity of merge sort is $O(n)$ due to the auxiliary space used to merge parts of the array.

Quick Sort | Runtime: $O(n \log(n))$ **average,** $O(n^2)$ **worst case. Memory:** $O(\log(n))$.

In quick sort, we pick a random element and partition the array, such that all numbers that are less than the partitioning element come before all elements that are greater than it. The partitioning can be performed efficiently through a series of swaps (see below).

If we repeatedly partition the array (and its sub-arrays) around an element, the array will eventually become sorted. However, as the partitioned element is not guaranteed to be the median (or anywhere near the median), our sorting could be very slow. This is the reason for the $O(n^2)$ worst case runtime.

```
1  void quickSort(int[] arr, int left, int right) {
2    int index = partition(arr, left, right);
3    if (left < index - 1) { // Sort left half
4      quickSort(arr, left, index - 1);
5    }
6    if (index < right) { // Sort right half
7      quickSort(arr, index, right);
8    }
9  }
10
11 int partition(int[] arr, int left, int right) {
12   int pivot = arr[(left + right) / 2]; // Pick pivot point
13   while (left <= right) {
14     // Find element on left that should be on right
15     while (arr[left] < pivot) left++;
16
17     // Find element on right that should be on left
18     while (arr[right] > pivot) right--;
19
20     // Swap elements, and move left and right indices
21     if (left <= right) {
22       swap(arr, left, right); // swaps elements
23       left++;
24       right--;
25     }
26   }
27   return left;
28 }
```

Radix Sort | Runtime: $O(kn)$ **(see below)**

Radix sort is a sorting algorithm for integers (and some other data types) that takes advantage of the fact that integers have a finite number of bits. In radix sort, we iterate through each digit of the number, grouping numbers by each digit. For example, if we have an array of integers, we might first sort by the first digit, so that the 0s are grouped together. Then, we sort each of these groupings by the next digit. We repeat this process sorting by each subsequent digit, until finally the whole array is sorted.

Unlike comparison sorting algorithms, which cannot perform better than $O(n \log(n))$ in the average case, radix sort has a runtime of $O(kn)$, where n is the number of elements and k is the number of passes of the sorting algorithm.

▸ Searching Algorithms

When we think of searching algorithms, we generally think of binary search. Indeed, this is a very useful algorithm to study.

In binary search, we look for an element x in a sorted array by first comparing x to the midpoint of the array. If x is less than the midpoint, then we search the left half of the array. If x is greater than the midpoint, then we search the right half of the array. We then repeat this process, treating the left and right halves as subarrays. Again, we compare x to the midpoint of this subarray and then search either its left or right side. We repeat this process until we either find x or the subarray has size 0.

Note that although the concept is fairly simple, getting all the details right is far more difficult than you might think. As you study the code below, pay attention to the plus ones and minus ones.

```
1  int binarySearch(int[] a, int x) {
2    int low = 0;
3    int high = a.length - 1;
4    int mid;
5
6    while (low <= high) {
7      mid = (low + high) / 2;
8      if (a[mid] < x) {
9        low = mid + 1;
10     } else if (a[mid] > x) {
11       high = mid - 1;
12     } else {
13       return mid;
14     }
15   }
16   return -1; // Error
17 }
18
19 int binarySearchRecursive(int[] a, int x, int low, int high) {
20   if (low > high) return -1; // Error
21
22   int mid = (low + high) / 2;
23   if (a[mid] < x) {
24     return binarySearchRecursive(a, x, mid + 1, high);
25   } else if (a[mid] > x) {
26     return binarySearchRecursive(a, x, low, mid - 1);
27   } else {
28     return mid;
29   }
30 }
```

Potential ways to search a data structure extend beyond binary search, and you would do best not to limit yourself to just this option. You might, for example, search for a node by leveraging a binary tree, or by using a hash table. Think beyond binary search!

Interview Questions

10.1 **Sorted Merge:** You are given two sorted arrays, A and B, where A has a large enough buffer at the end to hold B. Write a method to merge B into A in sorted order.

Hints: #332

pg 396

10.2 **Group Anagrams:** Write a method to sort an array of strings so that all the anagrams are next to each other.

Hints: #177, #182, #263, #342

pg 397

10.3 **Search in Rotated Array:** Given a sorted array of n integers that has been rotated an unknown number of times, write code to find an element in the array. You may assume that the array was originally sorted in increasing order.

EXAMPLE

Input: find 5 in {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}

Output: 8 (the index of 5 in the array)

Hints: #298, #310

pg 398

10.4 **Sorted Search, No Size:** You are given an array-like data structure `Listy` which lacks a size method. It does, however, have an `elementAt(i)` method that returns the element at index `i` in `O(1)` time. If `i` is beyond the bounds of the data structure, it returns `-1`. (For this reason, the data structure only supports positive integers.) Given a `Listy` which contains sorted, positive integers, find the index at which an element x occurs. If x occurs multiple times, you may return any index.

Hints: #320, #337, #348

pg 400

10.5 **Sparse Search:** Given a sorted array of strings that is interspersed with empty strings, write a method to find the location of a given string.

EXAMPLE

Input: `ball`, {"at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""}

Output: 4

Hints: #256

pg 401

10.6 **Sort Big File:** Imagine you have a 20 GB file with one string per line. Explain how you would sort the file.

Hints: #207

pg 402

10.7 **Missing Int:** Given an input file with four billion non-negative integers, provide an algorithm to generate an integer that is not contained in the file. Assume you have 1 GB of memory available for this task.

FOLLOW UP

What if you have only 10 MB of memory? Assume that all the values are distinct and we now have no more than one billion non-negative integers.

Hints: #235, #254, #281

pg 403

10.8 **Find Duplicates:** You have an array with all the numbers from 1 to N, where N is at most 32,000. The array may have duplicate entries and you do not know what N is. With only 4 kilobytes of memory available, how would you print all duplicate elements in the array?

Hints: #289, #315

pg 406

10.9 **Sorted Matrix Search:** Given an M x N matrix in which each row and each column is sorted in ascending order, write a method to find an element.

Hints: #193, #211, #229, #251, #266, #279, #288, #291, #303, #317, #330

pg 407

10.10 **Rank from Stream:** Imagine you are reading in a stream of integers. Periodically, you wish to be able to look up the rank of a number x (the number of values less than or equal to x). Implement the data structures and algorithms to support these operations. That is, implement the method `track(int x)`, which is called when each number is generated, and the method `getRankOfNumber(int x)`, which returns the number of values less than or equal to x (not including x itself).

EXAMPLE

Stream (in order of appearance): `5, 1, 4, 4, 5, 9, 7, 13, 3`

```
getRankOfNumber(1) = 0
getRankOfNumber(3) = 1
getRankOfNumber(4) = 3
```

Hints: #301, #376, #392

pg 412

10.11 **Peaks and Valleys**: In an array of integers, a "peak" is an element which is greater than or equal to the adjacent integers and a "valley" is an element which is less than or equal to the adjacent integers. For example, in the array {5, 8, 6, 2, 3, 4, 6}, {8, 6} are peaks and {5, 2} are valleys. Given an array of integers, sort the array into an alternating sequence of peaks and valleys.

EXAMPLE

Input: {5, 3, 1, 2, 3}

Output: {5, 1, 3, 2, 3}

Hints: #196, #219, #231, #253, #277, #292, #316

pg 414

Additional Questions: Arrays and Strings (#1.2), Recursion (#8.3), Moderate (#16.10, #16.16, #16.21, #16.24), Hard (#17.11, #17.26).

Hints start on page 662.