10. Sorting and Searching

A good approach is to run through the different sorting algorithms to see if one applies well to a situation.

E.g. Given a very large array of Person objects, sort the people in increasing order of age.

- It's a large array, so efficiency is very important
- 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 would be a perfect candidate for this algorithm. In face, we can make the buckets small (1 year each) and get O(n) running time.

Common Sorting Algorithms

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

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.

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

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 until array is sorted.

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

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 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.

```
void mergesort(int[] array){
  int[] helper = new int[array.length];
  mergesort(array, helper, 0, array.length - 1);
}

void mergesort(int[] array, int[] helper, int low, int high) {
  if (low < high) {
    int middle = (low + high) / 2;
    mergesort(array, helper, low, middle);
    mergesort(array, helper, middle+1, high);

    merge(array, helper, low, middle, high);
  }
}

void merge(int[] array, int[] helper, int low, int middle, int high) {
  // copy both halves into a helper array
  for (int i = low; i <= high; i++) {
    helper[i] = array[i];
  }

  int helperLeft = low;
  int helperRight = middle + 1;
  int current = low;

  // Iterate through helper array. Compare the left and right half, copying back
  // The smaller element from the two halves into the original array
  while (helperLeft <= middle && helperRight <= high) {
    if (helper[helperLeft] <= helper[helperRight]) {
      array[current] = helper[helperLeft];
      helperLeft++
    } else {
      array[current] = helper[helperRight];
      helperRight++;
    }
    current++;
  }
  // Copy the rest of the left side of the array into the target array
  int remaining = middle - helperLeft;
  for (int i = 0; i <= remaining; i++) {
    array[current + i] = helper[helperLeft + i];
  }
}
```

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 the 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.

If we repeatedly partition the array (and its sub-arrays) around an element, the array will eventually become sorted. As the partitioned element is not guaranteed to be the median, our sorting can be

very slow. This is the reason for the O(n^2) worst case runtime.

```
void quickSort(int[] array, int low, int high) {
  if (low < high) {
    // arr[partitionIndex] is now at the right place
    int partitionIndex = partition(arr, low, high);

    quickSort(arr, low, partitionIndex - 1);
    quickSort(arr, partitionIndex + 1, high);
  }
}

int partition(int[] arr, int low, int high) {
  int pivot = arr[low];
  int i = low; // i searches for element > pivot
  int j = high; // j searches for element <= pivot

  while (i < j){
    do {
      i++;
    } while (arr[i] <= pivot);
    do {
      j--;
    } while (arr[j] > pivot);
    swap(arr[i], arr[j]);
  }
  swap(arr[low], arr[j]);
  return j;
}
```

Radix Sort | Runtime: O(kn), where n is the number of elements and k is the number of passes of the sorting algorithm

In radix sort, we iterate through each digit of the number, grouping numbers by each digit. E.g. 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 the whole array is sorted.

Searching Algorithms

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 < midpoint, then we search the left half of the array. Else, search right half. Repeat this process, treating the left and right halves as subarrays. Again, we compare x to the midpoint of this subarray and search either left or right. Repeat this process until we either find x or the subarray size is 0.

```
int binarySearch(int[] a, int x) {
  int low = 0;
  int high = a.length - 1;
  int mid;

  while (low <= high) {
    mid = (low + high) / 2;
    if (a[mid] < x) {
      low = mid + 1;
    } else if (a[mid] > x) {
      high = mid - 1;
    } else {
      return mid;
    }
  }
  return -1;
}

int binarySearchRecursive(int[] a, int x, int low, int high) {
  if (low > high) return -1;

  int mid = (low + high) / 2;
  if (a[mid] < x) {
    return binarySearchRecursive(a, x, mid + 1, high);
  } else if (a[mid] > x) {
    return binarySearchRecursive(a, x, low, mid - 1);
  } else {
    return mid;
  }
}
```

Questions will be in another file