Sort

Sort

- Time Complexity
- Space Complexity
- Stable, unstable

Time Complexity

- O(n^2): Insertion sort, Bubble sort, Selection sort
- O(nlogn): QuickSort, MergeSort, HeapSort
- O(n): Couting Sort, Bucket Sort, Radix Sort

Time Complexity

We need to consider about the best and worst case

For example:

Best case for Bubble Sort is O(n)

Worst case for QuickSort is O(n^2)

Question: How the quicksort hit the worst case?

What if the partition is always 1:100000?

Time Complexity

The lower bound of **comparison sort** is O(nlogn)

Counting Sort, Bucket Sort and Radix Sort are not comparison sort.

To get O(n), you need to have some pre defined requirements for the data you want to sort.

Counting, Bucket and Radix

- These are not comparison sort
- you have some limits on the numbers you are going to sort
- Radix is just counting repeating multiple times

Space Complexity

- In-place and out-of-place
- in-place algorithm is an algorithm which transforms input using no auxiliary data structure

In-place: Bubble Sort, Insertion Sort, Selection Sort, QuickSort, HeapSort

Out-of place: Merge Sort, Counting Sort, Bucket Sort, Radix Sort

Stability

- stable sorting algorithms maintain the relative order of records with equal keys

Stable: MergeSort, Bubble Sort, Insertion Sort, Counting Sort, Bucket Sort, Radix Sort

Unstable: QuickSort, HeapSort, Selection Sort

Sort algorithm	Time complexity	Space complexity	Stability
Bubble	O(n^2)	O(1)	YES
Insertion	O(n^2)	O(1)	YES
Selection	O(n^2)	O(1)	NO
Quick	O(nlogn)	O(logn)	NO
Heap	O(nlogn)	O(1)	NO
Merge	O(nlogn)	O(n)	YES
Counting	O(n+k)	O(n+k)	YES
Bucket	O(n+k)	O(n+k)	YES
Radix	O(dn)	O(n)	YES

External Sorting

- The sort we talk about is internal sort
- External sorting is required when the data being sorted do not fit into the main memory of a computing device and instead they must reside in the slower external memory
- k-way merge
- Instead of only considering the sorting algorithm, you need to count for the I/O time

What do we need to know

- Some basic ideas on how these sorting algorithms work
- Arrays.sort and Collections.sort
- Comparator and Comparable

Largest Number

Given a list of non negative integers, arrange them such that they form the largest number.

For example, given [3, 30, 34, 5, 9], the largest formed number is 9534330.

Note: The result may be very large, so you need to return a string instead of an integer.

Largest Number

How to use sort to decide the largest number?

Example:

If we have 9 and 58. 958 > 589

-> To sort them, we need to concat them and compare which is larger.

-> This can apply to N numbers.

Largest Number

```
public String largestNumber(int[] nums) {
    if (nums == null || nums.length == 0) {
        return "";
    }
    String[] strs = new String[nums.length];
    for (int i = 0; i < nums.length; i++) {
        strs[i] = String.valueOf(nums[i]);
    }
    Arrays.sort(strs, new Comparator<String>(){
        public int compare(String str1, String str2) {
            return (str2 + str1).compareTo(str1 + str2);
        }
    });
    if (strs[0].charAt(0) == '0') {
        return "0";
    }
    StringBuilder result = new StringBuilder();
    for (int i = 0; i < strs.length; i++) {
        result.append(strs[i]);
    }
    return result.toString();
}
```

Best Meeting Point

A group of two or more people wants to meet and minimize the total travel distance. You are given a 2D grid of values 0 or 1, where each 1 marks the home of someone in the group. The distance is calculated using Manhattan Distance, where distance(p1, p2) = |p2.x - p1.x| + |p2.y - p1.y|.

For example, given three people living at (0,0), (0,4), and (2,2):

Result is (0,2)

```
1 - 0 - 0 - 0 - 1
|   |   |   |   |
0 - 0 - 0 - 0 - 0
|   |   |   |   |
0 - 0 - 1 - 0 - 0
```

Best Meeting Point

First we consider the one dimension case:

_____A_____B_____

_____A_____P_____B_____

_____A_____B___P___

As long as P is in the middle of A and B, the total distance is the same

_____A_____B_____P_____C_____D_____

The total distance is AD + BC as long as P is in the middle

Best Meeting Point

So For two dimension, since it is Manhattan Distance, we can just take x and y separately

Pair first and last, second and second last ...

If there are 2N points, then P is in the middle of the median two.

If there are 2N+1 points, then P is just the N+1 point

Best Meeting Point

```
public int minTotalDistance(int[][] grid) {
    List<Integer> ipos = new ArrayList<Integer>();
    List<Integer> jpos = new ArrayList<Integer>();
    for(int i = 0; i < grid.length; i++){
        for(int j = 0; j < grid[0].length; j++){
            if(grid[i][j] == 1){
                ipos.add(i);
                jpos.add(j);
            }
        }
    }
    int sum = 0;
    Collections.sort(ipos);
    Collections.sort(jpos);
    int i = 0, j = ipos.size() - 1;
    while (i < j) {
        sum += ipos.get(j) - ipos.get(i);
        sum += jpos.get(j) - jpos.get(i);
        i ++;
        j --;
    }
    return sum;
}
```

Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

For example,

Given [1,3],[2,6],[8,10],[15,18],

return [1,6],[8,10],[15,18].

Merge Intervals

We need to sort the intervals

Based on what? -> start

Go through all the results and merge them one by one

Merge Intervals

```
public class MyComparator implements Comparator<Interval> {
    @Override
    public int compare(Interval a, Interval b) {
        return a.start - b.start;
    }
}

public List<Interval> merge(List<Interval> intervals) {
    List<Interval> ans = new ArrayList<Interval>();
    if (intervals.size() == 0) return ans;

    Collections.sort(intervals, new MyComparator());

    int start = intervals.get(0).start;
    int end = intervals.get(0).end;

    for (int i = 0; i < intervals.size(); i++) {
        Interval inter = intervals.get(i);
        if (inter.start > end) {
            ans.add(new Interval(start, end));
            start = inter.start;
            end = inter.end;
        } else {
            end = Math.max(end, inter.end);
        }
    }
    ans.add(new Interval(start, end));

    return ans;
}
```

Merge Intervals

We do not sort but we can use TreeSet to help maintain the order

```
public List<Interval> merge(List<Interval> intervals) {
    TreeSet<Point> points = new TreeSet<Point>();
    List<Interval> results = new ArrayList<>();
    for (Interval interval: intervals) {
        points.add(new Point(interval.start, true));
        points.add(new Point(interval.end, false));
    }
    int start = 0, end = 0, count = 0;
    for (Point point: points) {
        if (point.start) {
            if (count == 0) start = point.x;
            count ++;
        } else {
            count --;
            if (count == 0) {
                end = point.x;
                results.add(new Interval(start, end));
            }
        }
    }
    return results;

}
```

Merge Intervals

```
class Point implements Comparable<Point> {
    final int x;
    final boolean start;
    private Point(int x, boolean start) {
        this.x = x;
        this.start = start;
    }
    public int compareTo(Point that) {
        if (this.x != that.x) {
            return this.x - that.x;
        } else if (this.start) {
            return -1;
        } else {
            return 1;
        }
    }
}
```

Maximum Gap

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Try to solve it in linear time/space.

Return 0 if the array contains less than 2 elements.

You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.

Maximum Gap

n numbers -> n buckets

gap for buckets -> (max - min) / n

then compare between each adjacent bucket

largest in one bucket with smallest in next bucket

All elements in the same bucket has no possibility to give the largest gap

Therefore, in each bucket, we only need to maintain the largest element and the smallest for the computation at the end

Maximum Gap

```java
public int maximumGap(int[] nums) {
    if (nums.length == 0) return 0;
    int max = nums[0];
    int min = nums[0];
    for(int num: nums) {
        max = Math.max(max, num);
        min = Math.min(min, num);
    }
    int len = (max - min) / nums.length + 1;
    List<List<Integer>> buckets = new ArrayList<>();
    int bucketSize = (max - min) / len + 1;
    for (int i = 0; i < bucketSize; i ++) {
        buckets.add(new ArrayList<Integer>());
    }
    for (int num: nums) {
        int i = (num - min) / len;
        if (buckets.get(i).isEmpty()) {
            buckets.get(i).add(num);
            buckets.get(i).add(num);
        } else {
            if (num < buckets.get(i).get(0)) buckets.get(i).set(0, num);
            if (num > buckets.get(i).get(1)) buckets.get(i).set(1, num);
        }
    }
    int gap = 0, prev = 0;
    for (int i = 1; i < bucketSize; i ++) {
        if (buckets.get(i).isEmpty()) continue;
        gap = Math.max(gap, buckets.get(i).get(0) - buckets.get(prev).get(1));
        prev = i;
    }
    return gap;
}
```

Homework

- H-Index
- Insert Interval
- Maximum Gap
- First Missing Positive
- Wiggle Sort II