CSM 61B
Fall 2020

61B Potpourri
Mentoring 13: November 16-20, 2020

1 Identify the Sort!

1.1 Each of the following sequences represent an array being sorted at some intermediate step. Match each sample with one of the sorting algorithms: **insertion sort, selection sort, heapsort, merge sort, quicksort**. The original array is below.

```
5103 9914 0608 3715 6035 2261 9797 7188 1163 4411
```

(a)
```
0608 5103 9914 3715 6035 2261 7188 9797 1163 4411
0608 3715 5103 6035 9914 2261 7188 9797 1163 4411
```

Merge sort

(b)
```
0608 1163 5103 3715 6035 2261 9797 7188 9914 4411
0608 1163 2261 3715 6035 5103 9797 7188 9914 4411
```

Selection sort

(c)
```
9797 7188 5103 4411 6035 2261 0608 3715 1163 9914
4411 3715 2261 0608 1163 5103 6035 7188 9797 9914
```

Heapsort

(d)
```
5103 0608 3715 2261 1163 4411 6035 9914 9797 7188
0608 2261 1163 3715 5103 4411 6035 9914 9797 7188
```

Quicksort

(e)
```
0608 5103 9914 3715 6035 2261 9797 7188 1163 4411
0608 2261 3715 5103 6035 9914 9797 7188 1163 4411
```

Insertion sort

2 Berkeley Bytes

2.1 For taxes, you must to submit a list of the ages of your customers in sorted order. Define `ageSort`, which takes an `int[] array` of all customers' ages and returns a sorted array. Assume customers are less than 150 years old.

```
public class BerkeleyBytes {
    private static int maxAge = 149;
    public static int[] histogram(int[] ages) {
        int[] ageCounts = new int[maxAge + 1];
        for (int age : ages) {
            ageCounts[age] += 1;
        }
        return ageCounts;
    }
    public static int[] ageSort(int[] ages) {
        int[] ageCounts = histogram(ages);
        int[] result = new int[ages.length];
        int index = 0;
        for (int age = 0; age < maxAge; age++) {
            for (int count = 0; count < ageCounts[age]; count++) {
                result[index] = age;
                index += 1;
            }
        }
        return result;
    }
}
```

2.2 Time passes and your restaurant is doing well. Unfortunately, our robot overlords advanced medicine to the point where humans are now immortal.

(a) How could we extend the algorithm to accept a list of any ages?

Radix sort. Sort the customers using the above algorithm, looking at only the last digit of their age. We would need 10 buckets, since the digit can only have 10 values. Repeat with the second to last digit, and so on, until the first digit sorted.

(b) When would we be able to use this type of sort?

The keys we want to sort must have some **base** (or radix). The type of item must be some combination of symbols.

3 Shortest Paths

3.1 Given a weighted, directed graph G where the weights of every edge in G are all integers between 1 and 10, and a starting vertex s in G, find the distance from s to every other vertex in the graph where the distance between two vertices is defined as the weight of the shortest path connecting them, or infinity if no such path exists.

(a) Design an algorithm for solving the problem better than Dijkstra's.

For every edge e in the graph, replace e with a chain of $w - 1$ vertices (where w is the weight of e) where the two ends of the chain are the endpoints of e.

Then run BFS on the modified graph, keeping track of the distance from v to each vertex from the original graph.

Alternatively, we can modify Dijkstra's algorithm. Since the runtime of Dijkstra's is bounded by the priority queue implementation, if we can come up with a faster priority queue, we can improve the runtime. Define our priority queue as an array of 11 linked list buckets. Keep track of a counter that represents our current position in the array. Each bucket corresponds to vertices of some distance from the start, s.

`removeMin()`: If the bucket with index counter is non-empty, remove and return the first vertex in its linked list. Otherwise, increment counter until we find a non-empty bucket. If the counter reaches 11, reset it to 0 and continue (so in effect our array is circular).

`insert()`: Given a vertex v and a distance d, insert the vertex into the beginning of the linked list at index $d \mod 11$.

This strategy works because the vertices we add to the priority queue at any time have a distance that is no more than 10 greater than the current distance.

(b) Give the runtime of your algorithm.

$\Theta(|V| + |E|)$ for running BFS on the modified graph, and $O(|V| + |E|)$ for modifying Dijkstra's priority queue.

4 Largest Perimeter

4.1 Given an array A of positive lengths, return the largest perimeter of a triangle with non-zero area, formed from 3 of these lengths. Recall the Triangle Inequality, which states that for any triangle, the sum of the lengths of any two sides must be greater than or equal to the length of the remaining side ($a + b > c$). If it is impossible to form any triangle of non-zero area, return 0.

For example, A = [2, 1, 2] returns 5. A = [1, 2, 1] returns 0. A = [3, 2, 3, 4] returns 10.

What is the runtime of your solution?

```
public int largestPerimeter(int[] A) {









}
```

Note: this problem was adapted from LeetCode (https://leetcode.com/problems/largest-perimeter-triangle/).

Note: this solution was adapted from LeetCode (https://leetcode.com/problems/largest-perimeter-triangle/solution/).

Without loss of generality, say the side lengths of the triangle are $a \leq b \leq c$. The necessary and sufficient condition for these lengths to form a triangle of non-zero area is $a + b > c$.

Say we knew c already. There is no reason not to choose the largest possible a and b from the array. If $a + b > c$, then it forms a triangle, otherwise it doesn't.

This leads to a simple algorithm: sort the array. For any c in the array, we choose the largest possible $a \leq b \leq c$: these are just the two values adjacent to c. If this forms a triangle, we return the answer. Else, we return 0.

Thus, we complete our function as follows:

```
public int largestPerimeter(int[] A) {
    Arrays.sort(A);
    for (int i = A.length - 3; i >= 0; --i) {
        if (A[i] + A[i+1] > A[i+2]) {
            return A[i] + A[i+1] + A[i+2];
        }
    }
    return 0;
}
```

This function takes $O(N \log N)$ time to sort A and $O(N)$ time to iterate through the for loop. Thus, the overall runtime is given by $O(N \log N)$.