**Union Find**

**public class** UF {  
 **private int**[] id; *//parents array* **private int**[] sz; *//record the size of the set which each vertex belongs to* **private int** count; *//the total number of set at a certain time* **public** UF(**int** n){  
 count = n;  
 id = **new int**[n];  
 sz = **new int**[n];  
 **for**(**int** i = 0; i < n; i++) {  
 id[i] = i;  
 sz[i] = 1;  
 }  
 }  
  
 **public int** getCount(){  
 **return** count;  
 }  
  
 **public boolean** connected(**int** p, **int** q){  
 **return** find(p) == find(q);  
 }  
  
 **public int** find(**int** p){  
 *//find the root of the set which p belongs to* **while**(p != id[p]) {  
 id[p] = id[id[p]];  
 p = id[p];  
 }  
 **return** p;  
 }  
  
 **public void** union(**int** p, **int** q){  
 **int** i = find(p);  
 **int** j = find(q);  
 **if**(i == j) **return**;  
 *//link the root with smaller size to the root with greater size* **if**(sz[i] < sz[j]){  
 id[i] = j;  
 sz[j] += sz[i];  
 }**else**{  
 id[j] = i;  
 sz[i] += sz[j];  
 }  
 count--;  
 }  
}

**Find Where To Seat**

**static class** Interval **implements** Comparable{  
 **public int start**;  
 **public int end**;  
 **public int length**;  
 **public int max**;  
  
 **public** Interval(**int** i, **int** j, **int** k){  
 **start** = i;  
 **end** = j;  
 **length** = j - i + 1;  
 **max** = k;  
  
 }  
 **public int** compareTo(Object o) {  
 Interval other = (Interval) o;  
 **if** (other.**length** > **length**) {  
 **return** 1;  
 } **else if** (other.**length** < **length**) {  
 **return** -1;  
 } **else** {  
 **if** (other.**start** == 0 || other.**end** == **max**) {  
 **return** 1;  
 } **else** {  
 **return** -1;  
 }  
 }  
 }  
}  
  
**static public void** allocate(Queue<Interval> q){  
 **if**(q.isEmpty()){  
 System.***out***.println(**"no seat!"**);  
 }**else** {  
 Interval temp = q.poll();  
 **if**(temp.**end** == temp.**max**){  
 System.***out***.printf(**"sit at index %d"**,temp.**end**);  
 System.***out***.println();  
 **if**(temp.**start** <= temp.**end** - 1) {  
 Interval left = **new** Interval(temp.**start**, temp.**end** - 1, temp.**max**);  
 q.add(left);  
 }  
 }**else**{  
 **int** res = (temp.**start** + temp.**end**) / 2;  
 System.***out***.printf(**"sit at index %d"**,res);  
 System.***out***.println();  
 **if**(temp.**start** <= res-1){  
 Interval left = **new** Interval(temp.**start**,res-1,temp.**max**);  
 q.add(left);  
 }  
 **if**(res+1 <= temp.**end**){  
 Interval right = **new** Interval(res+1,temp.**end**,temp.**max**);  
 q.add(right);  
 }  
 }  
 }  
}

Currency  
**public static double**[] calcEquation(String[][] equations, **double**[] values, String[][] queries) {  
 HashMap<String,LinkedList<String>> adjacent = **new** HashMap();  
 HashMap<String,LinkedList<Double>> weight = **new** HashMap();  
 *//build graph* **for**(**int** i = 0; i < equations.**length**; i++){  
 String[] equation = equations[i];  
 **if**(!adjacent.containsKey(equation[0])){  
 adjacent.put(equation[0],**new** LinkedList<String>());  
 weight.put(equation[0],**new** LinkedList<Double>());  
 }  
 **if**(!adjacent.containsKey(equation[1])){  
 adjacent.put(equation[1],**new** LinkedList<String>());  
 weight.put(equation[1],**new** LinkedList<Double>());  
 }  
 **double** value = values[i];  
 adjacent.get(equation[0]).add(equation[1]);  
 adjacent.get(equation[1]).add(equation[0]);  
 weight.get(equation[0]).add(value);  
 weight.get(equation[1]).add(1/value);  
 }  
 LinkedList<Double> res = **new** LinkedList<Double>();  
 **for**(**int** i = 0; i < queries.**length**; i++){  
 String start = queries[i][0], end = queries[i][1];  
 **if**(!*DFS*(adjacent,weight,start,end,**new** HashSet<String>(),res,1.0)){  
 res.add(-1.0);  
 }  
 }  
 **double**[] r = **new double**[queries.**length**];  
 **for**(**int** i = 0; i < res.size(); i++){  
 r[i] = res.get(i);  
 }  
 **return** r;  
}  
  
**public static boolean** DFS(HashMap<String,LinkedList<String>> adjacent, HashMap<String,LinkedList<Double>> weight, String start,String end, HashSet<String> set, LinkedList<Double> res, **double** temp){  
 **if**(start.equals(end)){  
 res.add(temp);  
 **return true**;  
 }  
 **if**(set.contains(start)){  
 **return false**;  
 }  
 **if**(!adjacent.containsKey(start)){  
 **return false**;  
 }  
 set.add(start);  
 **for**(**int** i = 0; i < adjacent.get(start).size(); i++){  
 **double** value = weight.get(start).get(i);  
 String next = adjacent.get(start).get(i);  
 **if**(*DFS*(adjacent,weight,next,end,set,res,temp\*value)){  
 **return true**;  
 }  
 }  
 set.remove(start);  
 **return false**;  
}

Topological Sorting



