Binary Index Tree

**static class** BIT {

**long**[] **tree**; **int**[] **data**; **public** BIT(**int**[] in){

**data**=in; **tree**=**new long**[**data**.**length**+1]; **for**(**int** i=1;i<**tree**.**length**;i++){

**int** index=i; **int** val=**data**[i-1]; **while**(index<**tree**.**length**){ **tree**[index]+=val; index+=Integer.*lowestOneBit*(index); }

} } *//sum from 0 to endIndex inclusive* **public long** rangeSum(**int** endIndex){

endIndex++; **long** sum=0; **while**(endIndex>0){

sum+=**tree**[endIndex]; endIndex-=Integer.*lowestOneBit*(endIndex); } **return** sum; } *//sum from l to r inclusive* **public long** rangeSum(**int** l, **int** r){

**return** rangeSum(r)-rangeSum(l-1); } **public void** updateDelta(**int** index, **int** delta){

**int** fenIndex=index+1; **int** val=delta; **data**[index]+=delta; **while**(fenIndex<**tree**.**length**){ **tree**[fenIndex]+=val; fenIndex+=Integer.*lowestOneBit*(fenIndex); } } **public void** updateVal(**int** index, **int** newVal){

**int** fenIndex=index+1; **int** val=newVal-**data**[index]; **data**[index]=newVal; **while**(fenIndex<**tree**.**length**){ **tree**[fenIndex]+=val; fenIndex+=Integer.*lowestOneBit*(fenIndex); } } **public** String toString(){

**return "Tree: "** + Arrays.*toString*(**tree**)+**"\n"**+**"Data:"** + Arrays.*toString*(**data**); }

}

Disjoint Set

**class** DisjointSet {

**int**[] **parent**; **int**[] **rank**; **public** DisjointSet(**int** size){ **parent**=**new int**[size]; **rank**=**new int**[size]; **for**(**int** i=0;i<size;i++)

**parent**[i]=i; } **public int** find(**int** x){ **if**(**parent**[x]==x)

**return** x; **else**

**return parent**[x]=find(**parent**[x]); } **public void** union(**int** x, **int** y){

**int** xRoot=find(x); **int** yRoot=find(y); **if** (**rank**[xRoot] < **rank**[yRoot])

**parent**[xRoot]=yRoot; **else if**(**rank**[yRoot] < **rank**[xRoot])

**parent**[yRoot]=xRoot; **else** { **parent**[yRoot]=xRoot;

**rank**[xRoot]++; }

}

}

Sparse Table

**class** SparseTable {

**int**[][] **table**; ArrayList<Integer> **pows**=**new** ArrayList<>(25); **public** SparseTable(**int**[] in){

**int** power=1; **int** exp=0; **pows**.add(power); **while**(power<in.**length**){

power<<=1; exp++; **pows**.add(power); } **table**=**new int**[in.**length**][exp]; **for**(**int** i=0;i<in.**length**;i++)

**table**[i][0]=in[i]; **for**(**int** j=1;j<exp;j++)

**for** (**int** i=0;i<in.**length**;i++){

**if**(i+**pows**.get(j-1)>=in.**length**) **table**[i][j]=**table**[i][j-1]; **else**

**table**[i][j]=Math.*min*(**table**[i][j-1],**table**[i+**pows**.get(j-1)][j-1]); } } *//minimum from l to r inclusive* **public int** rmq(**int** left, **int** right){

**if**(left==right)

**return table**[left][0]; **if**(left==right-1)

**return table**[left][1];

**int** diff=right-left+1; **int** exp=0; **while**(**pows**.get(exp)<diff)

exp++; exp--; **return** Math.*min*(**table**[left][exp],**table**[right-**pows**.get(exp)+1][exp]); } }

TopSort

**static class** topSort{

HashMap<Integer,Integer> **indegree**=**new** HashMap<>(); ArrayList<Integer> **ordering**=**new** ArrayList<>(); *//Assumes all vertices in the edges keyset* **public** topSort(HashMap<Integer,HashSet<Integer>> edges){

**for**(**int** key:edges.keySet())

**indegree**.put(key,0); **for**(**int** key:edges.keySet())

**for**(**int** out:edges.get(key))

**indegree**.put(out,**indegree**.remove(out)+1);

ArrayDeque<Integer> deque=**new** ArrayDeque<>(); **for**(**int** key:edges.keySet())

**if**(**indegree**.get(key)==0)

deque.add(key);

**while**(!deque.isEmpty()){

**int** cur=deque.pollFirst(); **ordering**.add(cur); **for**(**int** out:edges.get(cur)) {

**int** val=**indegree**.get(out); **indegree**.put(out,val-1); **if**(val-1==0)

deque.add(out); }

} }

}

KnapSack

**class** KnapSack {

**static int**[][] *dp*; **static int**[] *weight*={}; **static int**[] *vals*={}; **static int** *maxWeight*=11; **public static int** knapSack(**int** index, **int** curWeight){

**if**(index>=*vals*.**length**)

**return** 0; **if**(*dp*[index][curWeight]!=0)

**return** *dp*[index][curWeight]; **if**(curWeight+*weight*[index]>*maxWeight*)

**return** *dp*[index][curWeight]=*knapSack*(index+1,curWeight); **else**

**return** *dp*[index][curWeight]=Math.*max*(*knapSack*(index+1,curWeight),*vals*[index]+*knapSack*(index+1 ,curWeight+*weight*[index]));

} **public static void** main(String args[]){

*dp*=**new int**[*weight*.**length**][*maxWeight*+1]; System.***out***.println(*knapSack*(0,0)); } }

Inversions

**class** Inversions{

**int**[] **list**; **int count**; **public** Inversions(**int**[] in){

**list**=in.clone(); Arrays.*sort*(in); HashMap<Integer,Integer> compressMap=**new** HashMap<>(); **for**(**int** i=0;i<in.**length**;i++){ compressMap.put(in[i],i); } **for**(**int** i=0;i<**list**.**length**;i++)

**list**[i]=compressMap.get(**list**[i]); **int**[] empty=**new int**[in.**length**]; BIT fenwick=**new** BIT(empty); **for**(**int** i=0;i<**list**.**length**;i++){

**count**+=fenwick.rangeSum(**list**[i],**list**.**length**-1); fenwick.updateDelta(**list**[i],1); } } }

Strongly Connected Components

**class** connectedComponents {

ArrayList<ArrayList<Integer>> **groups**=**new** ArrayList<>(); ArrayList<Integer> **curGroup**=**new** ArrayList<>(); ArrayDeque<Integer> **stack**=**new** ArrayDeque<>(); HashSet<Integer> **visited**=**new** HashSet<>(); HashMap<Integer, HashSet<Integer>> **edges**; HashMap<Integer, HashSet<Integer>> **reverse**; **public** connectedComponents(HashMap<Integer, HashSet<Integer>> edges,HashMap<Integer, HashSet<Integer>> reverse){

**this**.**edges**=edges; **this**.**reverse**=reverse; **for**(**int** start: edges.keySet())

DFSorder(start); **visited**=**new** HashSet<>(); *// System.out.println(stack);* **while**(!**stack**.isEmpty()){

DFSgroup(**stack**.pollFirst()); **if**(**curGroup**.size()!=0) {

**groups**.add(**curGroup**); **curGroup**=**new** ArrayList<>(); } } } **public void** DFSorder(**int** start){

**if**(**visited**.contains(start))

**return**; **visited**.add(start); **for**(**int** e:**edges**.get(start))

DFSorder(e); **stack**.addFirst(start);

} **public void** DFSgroup(**int** start){

**if**(**visited**.contains(start))

**return**; **visited**.add(start); **curGroup**.add(start); **for**(**int** e:**reverse**.get(start))

DFSgroup(e); } }

Totient Sieve

**class** totientSieve {

*//calculate totient values from 1 to phiValues.length-1* **public** totientSieve(**int**[] phiValues){

**for** (**int** i = 1; i < phiValues.**length**; i++)

phiValues[i] = i;

**for** (**int** p = 2; p <phiValues.**length**; p++) {

*//p is prime* **if** (phiValues[p] == p) {

phiValues[p] = p - 1; *// Update multiples of p* **for** (**int** i = 2 \* p; i < phiValues.**length**; i += p)

phiValues[i] = (phiValues[i] / p) \* (p - 1); } } } }

Letter Ordering

**class** letterOrdering {

**public static void** main(String[] args) **throws** IOException {

BufferedReader br=**new** BufferedReader(**new** InputStreamReader(System.***in***)); **int** len=Integer.*parseInt*(br.readLine()); ArrayList<String> list=**new** ArrayList<>(len); **for**(**int** i=0;i<len;i++)

list.add(br.readLine()); HashMap<Integer,HashSet<Integer>> edges=**new** HashMap<>(); **int** start=(**int**)**'a'**; **for**(**int** i=start;i<start+26;i++){

edges.put(i,**new** HashSet<>()); } **for**(**int** i=0;i<len-1;i++){

String cur=list.get(i); String next=list.get(i+1); **int** index=0; **while**(cur.charAt(index)==next.charAt(index)){

index++; **if**(index==cur.length()||index==next.length()) {

**if**(next.length()<cur.length()){

System.***out***.println(**"Impossible"**); **return**; } **break**; } } **if**(index<Math.*min*(cur.length(),next.length())){

edges.get((**int**)cur.charAt(index)).add((**int**)next.charAt(index)); } } *// System.out.println(edges);*

topSort sort=**new** topSort(edges); String out=**""**; **for**(**int** e:sort.**ordering**)

out+=((**char**)e); System.***out***.println(out.length()==26?out:**"Impossible"**); } }