HashMap (time complexity, hash->array, hash->arrayList)

Single List/Double linked list (Iterator, remove, insert)

Stack/Queue/PriorityQueue /Heap

Tree

(BST, pre-order, inorder, post order, dfs, bfs, depth, balanced tree, full tree)

Bit manipulation

(most significant bit, least significant bit)

Search – Binary search

Sort

(merge sort (Divid and Conquer), insertion sort)

Graph

(vector graph, non-vector graph, dfs/bfs iterate)

Backtrack

(base condition, recursion)

DP

(数组型，序列型，划分型， 背包， 贪心， 博弈， 除博弈从后向前找规律)

Math

(坐标， 距离， 向量， 斜率， 多边形)

Two pointers

(sliding windows)

Union find

Java built in functions

**Data Structures and Operations**

**Charcter**:

Character.isDigit()

**Array:**

题目类型： (找规律)

**String:**

charAt/substring(startIndex, endIndex)

**StringBuilder:**

Append/charAt/deleteCharAt/reverse/insert(I, c)/indexOf(string/c)/setCharAt(I, c)

**ArrayList:**

Insert(index, value), add(), size(), remove()//o(n) time

**Single Linked List:**

**Gg: 369 plus one, 23 merge k sorted list, 382 linked list random node, 379 design phone dic**

**Delete:**

**public** ListNode removeElements(ListNode head, **int** val) {

**if**(head==**null**) **return** **null**;

ListNode dummy = **new** ListNode(0);

dummy.next = head;

ListNode pre = dummy;

ListNode cur = head;

**while**(cur!=**null**){

**if**(cur.val == val){

pre.next = cur.next;

}**else**{

pre = pre.next;

}

cur = cur.next;

}

**return** dummy.next;

}

203. Remove linked list elements

**Reverse:**

Method1: recursive (on space, on time)

**public** ListNode reverseList(ListNode head) {

**return** reverseList(head, **null**);

}

**private** ListNode reverseList(ListNode head, ListNode newHead){

**if**(head==**null**) **return** newHead;

ListNode next = head.next;

head.next = newHead;

**return** reverseList(next, head);

}

Method2: (o(1) space, o(n) time)

**public** ListNode reverseList(ListNode head) {

**if**(head == **null**) **return** **null**;

ListNode newHead = **null**;

**while**(head!=**null**){

ListNode next = head.next;

head.next = newHead;

newHead = head;

head = next;

}

**return** newHead;

}

reverse ([m,n]):

用一个dummy node 来记录开始以及“前一个”

**public** ListNode reverseBetween(ListNode head, **int** m, **int** n) {

ListNode dummy = **new** ListNode(0);

dummy.next = head;

ListNode pre = dummy;

//1->2->3->4->5

**for**(**int** i=0;i<m-1;i++){

pre = pre.next;

}

ListNode start = pre.next;

ListNode then = start.next;

**for**(**int** i=0;i<n-m;i++){

start.next = then.next;

then.next = pre.next;

pre.next = then;

then = start.next;

}

**return** dummy.next;

}

92.Reverse Linked List II

**Merge**

(k sorted lists)

**public** ListNode mergeKLists1(List<ListNode> lists) {

**if**(lists==**null** || lists.size()==0) **return** **null**;

ListNode Dummy = **new** ListNode(0);

PriorityQueue<ListNode> pq = **new** PriorityQueue<>(lists.size(),

**new** Comparator<ListNode>(){

@Override

**public** **int** compare(ListNode l1, ListNode l2){

**return** l1.val - l2.val;

}

});

**for**(ListNode l:lists){

**if**(l!=**null**)

pq.offer(l);

}

ListNode head = Dummy;

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

ListNode node = pq.poll();

**if**(node.next!=**null**){

pq.offer(node.next);

}

head.next = node;

head = head.next;

}

**return** Dummy.next;

}

Sort: (insertation sort)

**public** ListNode insertionSortList(ListNode head) {

**if**(head == **null**) **return** head;

ListNode dummy = **new** ListNode(0);

//dummy.next = head;

ListNode pre = dummy;//start of the sorted list

ListNode cur = head;//node need to be moved

**while**(cur!=**null**){

ListNode next = cur.next;//next node will be inserted

**while**(pre.next!=**null** && pre.next.val<cur.val){

//find place to insert

pre = pre.next;

}

//insert cur to pre and pre.next

cur.next = pre.next;

pre.next = cur;

cur = next;

pre = dummy;

}

**return** dummy.next;

}

147. insertation sort

**Double Linked List:**

**Stack:**

**Queue:**

**PriorityQueue:**

**Heap/Deque:**

**HashSet:**

keySet() / values()/ add/ put/ getOrDefault

**TreeSet:**

Same with HashSet, only contains unique value.

Operations:

addAll/ add/ Ceiling closest >= / floor closest <= / higher closest > / lower closest < / headset ( o, inclusive) <(=) smaller objects / tailSet (0, inclusive(true/fase)) >(=) larger objects/first/ last

QS: Max Sum of Rectangle No Larger Than K – 363

//find the max subarray no more than k ->O(nlogn) time

TreeSet<Integer> set = **new** TreeSet<>();

set.add(0);

**int** curSum =0;

**for**(**int** l=0;l<m;l++){

curSum += col[l];

Integer ceiling = set.ceiling(curSum-k);

//System.out.println(curSum + ", c:" + ceiling);

**if**(ceiling!=**null**){

maxSum = Math.*max*(maxSum, curSum-ceiling);

}

set.add(curSum);

}

**Tree:**

Iterator:

Remove node:

Insert node:

Bst:

Trie Tree:

**Graph:**

对于图论的题， 首先找到一个合适的数据结构来存储图（adjacency hashmap/matrix）, 然后找到一个合适的方式来遍历/检索图。 BFS/DFS/Topological sort（查loop）/Union find（UF针对多少个圈）

DFS: 首先考虑base condition, 其次如何存储visited, 再次如何进行下层搜索，最后返回要返回的值（error condition/ result）Time: O(deepth)

BFS: 首先要准备好个queue, 然后初始化时选好起点（特定点/边界），然后进行进Q， 出Q的逐层搜索。对于有权值得路径，可以用priorityQueue. O(deepth)

Topological sort: 主要用途scheduling job/pre-requisite 问题, 老爸节点先遍历，也可用来查图中有没有loop。 TIME O(V+E)

Graph Initial:

List/HashMap:

HashMap<String, HashMap<String, Double>> map = new HashMap<>();// has weight

HashMap<String, HashMap<String>> map = new HashMap<>();

HashMap<String, List<String>> map = new HashMap<>();

Adjacency matrix matrix [i][j] // -> I, j has edge

Iterate:

DFS：

**public** **double**[] calcEquation1(String[][] equations, **double**[] values, String[][] queries) {

**if**(equations==**null** || equations.length==0)

**return** **new** **double** [] {};

HashMap<String, HashMap<String, Double>> map = **new** HashMap<>();

**int** n = equations.length;

**for**(**int** i=0;i<n;i++){//initial map

String a = equations[i][0];

String b = equations[i][1];

**double** val = values[i];

**if**(!map.containsKey(a)){

map.put(a, **new** HashMap<>());

}

**if**(!map.containsKey(b)){

map.put(b, **new** HashMap<>());

}

map.get(a).put(b, val);

**if**(val!=0)

map.get(b).put(a, 1/val);

}

**int** m = queries.length;

**double** [] res = **new** **double** [m];

**for**(**int** i=0;i<m;i++){

Double r = dfs(map, queries[i][0], queries[i][1], **new** HashSet<>());

res[i] = r==**null**?-1:r;

}

**return** res;

}

**public** Double dfs(HashMap<String, HashMap<String, Double>> map, String s, String e

, HashSet<String> visited){

**if**(!map.containsKey(s)){//base condition

**return** **null**;

}

**if**(visited.contains(s+":"+e)) {

**return** **null**;

}

**if**(s.equals(e)) **return** 1.0;

HashMap<String, Double> values = map.get(s);

visited.add(s+":"+e);//set visited

**for**(String k: values.keySet()){//do things

Double res = dfs(map, k, e, visited);

**if**(res!=**null**){

**return** res\*map.get(s).get(k);

}

}

visited.remove(s+":"+e);//reset visited

**return** **null**;//return result

}

用priorityQueue 让自己的子节点有顺序

**public** List<String> findItinerary(String[][] tickets) {

List<String> res = **new** ArrayList<>();

**if**(tickets==**null** || tickets.length==0 || tickets[0].length==0) **return** res;

HashMap<String, PriorityQueue<String>> map = **new** HashMap<>();

**for**(String [] t:tickets){//initial map

String from = t[0];

String to = t[1];

**if**(!map.containsKey(from)){

map.put(from, **new** PriorityQueue<>());

}

map.get(from).offer(to);

}

dfs(map, res, "JFK");//start

**return** res;

}

**private** **void** dfs(HashMap<String, PriorityQueue<String>> map, List<String> res, String from){

**if**(map.containsKey(from)){

PriorityQueue<String> pq = map.get(from);

**while**(pq!=**null** && !pq.isEmpty()){

dfs(map, res, pq.poll());//don’t need to recover as polled

}

}

res.add(0, from);

}

BFS：

Topological Sort:

用一个indegree 的hashmap 来存储有多少node 指向这个node. 当这个node 前所有node遍历结束， 将这个node 推入queue中。Topologicial sort 解决一个序列问题。

Clone Graph:

**public** String alienOrder1(String[] words) {

Map<Character, Set<Character>> map=**new** HashMap<Character, Set<Character>>();

Map<Character, Integer> degree=**new** HashMap<Character, Integer>();

String result="";

**if**(words==**null** || words.length==0) **return** result;

**for**(String s: words){

**for**(**char** c: s.toCharArray()){

degree.put(c,0);

}

}

**for**(**int** i=0; i<words.length-1; i++){

String cur=words[i];

String next=words[i+1];

**int** length=Math.*min*(cur.length(), next.length());

**for**(**int** j=0; j<length; j++){

**char** c1=cur.charAt(j);

**char** c2=next.charAt(j);

**if**(c1!=c2){

Set<Character> set=**new** HashSet<Character>();

**if**(map.containsKey(c1)) set=map.get(c1);

**if**(!set.contains(c2)){

set.add(c2);

map.put(c1, set);

degree.put(c2, degree.get(c2)+1);

}

**break**;

}

}

}

Queue<Character> q=**new** LinkedList<Character>();

**for**(**char** c: degree.keySet()){

**if**(degree.get(c)==0) q.add(c);

}

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

**char** c=q.remove();

result+=c;

**if**(map.containsKey(c)){

**for**(**char** c2: map.get(c)){

degree.put(c2,degree.get(c2)-1);

**if**(degree.get(c2)==0) q.add(c2);

}

}

}

**if**(result.length()!=degree.size()) **return** "";

**return** result;

}

DFS:

HashMap<Integer, UndirectedGraphNode> map = **new** HashMap<>();//visited

**public** UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

**if**(node==**null**) **return** node;

**if**(map.containsKey(node.label)) **return** map.get(node.label);

UndirectedGraphNode cur = **new** UndirectedGraphNode(node.label);

map.put(node.label, cur);

**for**(UndirectedGraphNode n:node.neighbors){

cur.neighbors.add(cloneGraph(n));

}

**return** cur;

}

BFS:

**public** UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

**if**(node==**null**) **return** node;

HashMap<Integer, UndirectedGraphNode> map = **new** HashMap<>();

Queue<UndirectedGraphNode> queue = **new** LinkedList<>();

queue.offer(node);//initial the queue

UndirectedGraphNode res = **new** UndirectedGraphNode(node.label);

map.put(node.label, res);

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

UndirectedGraphNode n = queue.poll();

**for**(UndirectedGraphNode neighbor: n.neighbors){

//create each node and save into map

**if**(!map.containsKey(neighbor.label)){//avoid infinite loop

UndirectedGraphNode cur

= **new** UndirectedGraphNode(neighbor.label);

map.put(cur.label, cur);

queue.offer(neighbor);

}

//add

map.get(n.label).neighbors.add(map.get(neighbor.label));

}

}

**return** res;

}

**Union Find：**分组, 查找

1. In Undirect graph detect cycle

(union by rank and path compression – log(n) time)

Check adjecent path: (2d union)

**public** List<Integer> numIslands21(**int** m, **int** n, **int**[][] positions) {

List<Integer> list = **new** ArrayList<>();

**if**(positions==**null** || positions.length==0) **return** list;

Union un = **new** Union(m, n);//initial union map

**int** [][] dirs = **new** **int** [][] {{0,-1},{-1,0},{1,0},{0,1}};

//add each node to the union, then search 4 directions check if we can merge it

**for**(**int** [] pos: positions){

**int** x = pos[0], y = pos[1];

**int** p = un.add(x, y);//add into un

**for**(**int** [] d: dirs){

**int** q = un.getId(x+d[0], y+d[1]);//check existed of current node

**if**(q>0 && !un.find(p, q)){

un.unite(p,q);

}

}

list.add(un.getCount());

}

**return** list;

}

**class** Union{

**public** **int** [] parents;

**int** [] ranks;

**int** m, n, count;

**public** Union(**int** m, **int** n){

**this**.m = m;

**this**.n = n;

**this**.parents = **new** **int** [m\*n+1];

**this**.ranks = **new** **int** [m\*n+1];

**this**.count = 0;

}

**public** **int** getCount(){

**return** count;

}

**public** **int** index(**int** x, **int** y){//compress 2d vector

**return** x\*n+y+1;

}

**public** **int** getId(**int** x, **int** y){

**if**(x>=0 && x<m && y>=0 && y<n)

**return** parents[index(x, y)];

**else**

**return** 0;

}

**public** **int** add(**int** x, **int** y){

**int** i = index(x, y);

parents[i] = i;

ranks[i] = 1;

count++;

**return** i;

}

**public** **boolean** find(**int** p, **int** q){

**return** root(p) == root(q);

}

**public** **void** unite(**int** p, **int** q){

**int** i = root(p), j = root(q);

**if**(ranks[i]<ranks[j]){//fast unite by ranks/weight

parents[i] = j;

ranks[j] += ranks[i];

}**else**{

parents[j] = i;

ranks[i] += ranks[j];

}

count--;

}

**public** **int** root(**int** i){

**for**(;i!=parents[i]; i=parents[i]){

parents[i] = parents[parents[i]];//compression tree structure

}

**return** i;

}

}

305. number of islands

**NP-Complete:**

Java Build in functions:

(comparator, sort, binary search)

**Algorithms (use data structure wisely):**

Sort:

Merge Sort:

Search:

Binary search:

Two pointers/Sliding Windows:

KMP:

Virtual indexing:

Recursive/Backtracking:

Dynamic programming (back page/gready):

**Math:**

1. 多边形 (Convex 凸/ concave 凹 polygon)

Check cross product/anti-clock/clock wise

//check the second line is clock wise or anti-clock wise

**private** **int** check(Point p1, Point p2, Point p3){

**return** (p2.y-p1.y)\*(p3.x-p2.x)-(p3.y-p2.y)\*(p2.x-p1.x);

}

If(check()>0) clock wise

QS: Erect the Fence l-578/ Convex Polygon l-469 (Hash, Math)

**Reservoir sampling:**

Gg:382. Linked List Random Node

http://blog.jobbole.com/42550/

(在数据流中随机取出数， 使每个被抽到的概率相同)

When random(n+1) == n, it means it’s 1/n possibility for each node now.

**public** **int** getRandom() {

//when random(n) == n, currently all node has the same possiblilty 1/n

ListNode cur = head;

**int** res = cur.val;

**for**(**int** i=1;cur.next!=**null**;i++){

cur = cur.next;

**if**(rand.nextInt(i+1) == i) res = cur.val;

}

**return** res;

}

**Topic:**

**Palindrome:**

**judge palindrome:**

**find all palindrome array in a string:**

Reminder:

1. 对于数组型， 和数字型的题目， 考虑INTEGER.MAX/INTEGER.MIN