# Find duplicate files

class Solution {

HashSet<String > visited = new HashSet<>();

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

HashMap<Integer, HashMap<Integer, List<String>>> mm = new HashMap<>(); // hash with size first, then file content.

boolean isDir(String path)  
List<String> listDir(String path)  
String joinPath(String path, String subPath)

int getSize(Sting file);

String getContent(String file);

Int SHA512(String content);

isSymbolicLink()

isHardLink()

private void addFileToMap(String f){

// is a file, add to visited or skip.

If(!visited.contains(f)) return;

//need to check how to get the content of the file

String[] output = f.split(" ");

String p = output[0] + "/";

for (int i = 1; i < output.length; i++){

String[] text = output[i].split("\\(");

String name = text[0];

String content = text[1].substring(0, text[1].length()-1);

Or

String content = getContent(f);

Int size = getSize(f);

Visited.add(f);

List<String> l; // list of same content files.

If(mm.get(size) != null){

HashMap<Integer, List<String>> m = mm.get(size);

cHash = SHA512(content); // compare byte by byte

if(m.get(cHash) != null){

l = mm.get(cHash);

}else{

l = new ArrayList<String>();

}

}

l.add(f);

m.put(cHash, l);

}

//bfs

Queue<int> q;

q.push(initial);

while (!q.empty())

{

Int cur = q.front(); q.pop();

}

public List<List<String>> findDuplicate(String paths) { //bfs

if(!isDir(paths)) return null;

LinkedList<String> queue = new LinkedList<>();

queue.add(paths);

While(queue.size() > 0){

String cur = queue.poll();

List<String> subDir = listDir(cur);

For(String f : subDir){

f = joinPath(cur, f); // need to check with interviewer, can join both file and folder?

// isDir check a full path or just the sub-path?

If (!isDir(f)){

addFileToMap(f);

}else{

// is a directory, keep searching. Assume directory will not contain cycle(need to check, only symbolic links exist, A -> B -> C -> A doesn’t exist.) Bfs idea

Queue.add(f);

}

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

Iterator it2 = mm.keySet().iterator();

while(it2.hasNext()){

Object key = it2.next();

if (mm.get(key).size() > 1)

res.add(mm.get(key));

}

return res;

}

}

**Bfs**

class Graph

{

    private int V;   // No. of vertices

    private LinkedList<Integer> adj[]; //Adjacency Lists

    // Constructor

    Graph(int v)

    {

        V = v;

        adj = new LinkedList[v];

        for (int i=0; i<v; ++i)

            adj[i] = new LinkedList();

    }

    // Function to add an edge into the graph

    void addEdge(int v,int w)

    {

        adj[v].add(w);

    }

 // prints BFS traversal from a given source s

    void BFS(int s)

    {

        // Mark all the vertices as not visited(By default

        // set as false)

        boolean visited[] = new boolean[V];

        // Create a queue for BFS

        LinkedList<Integer> queue = new LinkedList<Integer>();

        // Mark the current node as visited and enqueue it

        visited[s]=true;

        queue.add(s);

        while (queue.size() != 0)

        {

            // Dequeue a vertex from queue and print it

            s = queue.poll();

            System.out.print(s+" ");

            // Get all adjacent vertices of the dequeued vertex s

            // If a adjacent has not been visited, then mark it

            // visited and enqueue it

            Iterator<Integer> i = adj[s].listIterator();

            while (i.hasNext())

            {

                int n = i.next();

                if (!visited[n])

                {

                    visited[n] = true;

                    queue.add(n);

                }

            }

        }

    }

    // Driver method to

    public static void main(String args[])

    {

        Graph g = new Graph(4);

        g.addEdge(0, 1);

        g.addEdge(0, 2);

        g.addEdge(1, 2);

        g.addEdge(2, 0);

        g.addEdge(2, 3);

        g.addEdge(3, 3);

        System.out.println("Following is Breadth First Traversal "+

                           "(starting from vertex 2)");

        g.BFS(2);

    }

}

**Follow up**

1.速度太慢怎么办。 比size

2.md5 collision怎么办。sha256，或者byte by byte

3.最慢的part是哪里？ i/o

**1. Imagine you are given a real file system, how will you search files? DFS or BFS ?**  
In general, BFS will use more memory then DFS. However BFS can take advantage of the locality of files in inside directories, and therefore will probably be faster

Could your directories contain links and would you traverse the links? In that case, is it possible for the links to make a loop? In such a case, BFS makes more sense if you want to ignore the cycle-checking. Otherwise, it makes no difference.

How is the distribution of errors? Could it be that one directory contains most errors while others are almost empty of errors? In that case BFS is more likely to end sooner because it searches all directories little by little. In such a case, you would spend a long time with DFS in one huge directory tree that contains say 1 error in the very bottom leaves only to find out the next directory contained all the bugs you need right at level 1. If the errors are distributed more evenly, again, it doesn't matter what you use.

How big is your structure? If you have a tree with branching factor n (n subdirectories per each directory) and the tree has depth d, the BFS could take O(n^d) memory, while DFS could be written in such a way that it takes only O(d) memory (or in a simpler implementation O(d\*n)) which in real huge directories could make a difference.

**2. If the file content is very large (GB level), how will you modify your solution?**  
In a real life solution we will not hash the entire file content, since it's not practical. Instead we will first map all the files according to size. Files with different sizes are guaranteed to be different. We will than hash a small part of the files with equal sizes (using MD5 for example). Only if the md5 is the same, we will compare the files byte by byte

**3. If you can only read the file by 1kb each time, how will you modify your solution?**  
This won't change the solution. We can create the hash from the 1kb chunks, and then read the entire file if a full byte by byte comparison is required.

Rolling hash: compare only the last chunks.

Multi-IO

**4. What is the time complexity of your modified solution? What is the most time consuming part and memory consuming part of it? How to optimize?**  
Time complexity is O(n^2 \* k) since in worse case we might need to compare every file to all others. k is the file size

**5. How to make sure the duplicated files you find are not false positive?**  
We will use several filters to compare: File size, Hash and byte by byte comparisons.

Soft links is a special kind of file that points to another file, much like a shortcut. Unlike a hard link, a symbolic link does not contain the data in the target file. This difference gives symbolic links certain qualities that hard links do not have, such as the ability to link to directories, or to files on remote computers networked through NFS. Also, when you delete a target file, symbolic links to that file become unusable, whereas hard links preserve the contents of the file.

<https://medium.com/@wendymayorgasegura/what-is-the-difference-between-a-hard-link-and-a-symbolic-link-8c0493041b62>

ignore symbolic link or

HashSet<String> visited = new HashSet<>();

if(symbolic link, trace to the source file, and add the unvisited source file)

visted.add(l);

**SHA-2** (**Secure Hash Algorithm 2**) is a set of [cryptographic hash functions](https://en.wikipedia.org/wiki/Cryptographic_hash_function) SHA 512. results in a **128-digit hexadecimal number** that is highly unlikely to match the value produced for a different input.

# Game of life

00

01

10

11

class Solution {

private int check(int[][] board, int row, int col, int x, int y){

int ret = 0;

for(int i = Math.max(0, x - 1); i <= Math.min(row - 1, x + 1); i++){

for (int j = Math.max(0, y - 1); j <= Math.min(col - 1, y + 1); j++){

ret += board[i][j] & 1;

}

}

ret -= board[x][y] & 1;

return ret;

}

public void gameOfLife(int[][] board) {

int m = board.length;

int n = board[0].length;

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

for (int j = 0 ; j < n; j++){

int live = check(board, m, n, i, j);

if (live == 3 && board[i][j] == 0) board[i][j] = 2;

if (live >= 2 && live <= 3 && board[i][j] == 1) board[i][j] = 3;

}

}

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

for (int j = 0 ; j < n; j++){

board[i][j] >>= 1;

}

}

}

}

Follow up:

1.follow up的时候会问时间空间复杂度 会问的很细到 比如1m\*1m 大小的grid 空间复杂度会占用多少个bytes 楼主好久不做运算突然间算一下感觉还是有点生疏的 125000000000 = 1.25 \*10^9

2.一开始没有直接给inline的转化方法用的是最土的存到另一个数组里 后来居然

也没有让优化

3.follow up优化3行3行的读会要求写代码。提前练习一下准备好就行了 不用跑

class Solution {

// API

int[] read\_row(BoardMetadata board, int row\_id); // given a board and a row\_id, return the correspoding row

void write\_row(BoardMetadata board, int row\_id, const int[] to\_write); // given a board, a row\_id and a row content, write the row content to board at row\_id

public void gameOfLife(BoardMetadata board)

{

int n = board.width;

/\*

cache 0 1 2

board 3 1 2

cache[row%3] = read\_row(board, row);

\*/

int m = board.height;

int[] cache = new int[3];

for (int i = 0; i < 2; ++i) cache[i] = read\_row(board, i);

for (int row = 0; row < m; ++row)

{

for (int col = 0; col < n; ++col)

{

int live = check\_cache(cache, n, m, row, j);

if (live == 3 && cache[row%3][j] == 0) cache[row%3][j] = 2;

if (live >= 2 && live <= 3 && cache[row%3][j] == 1) cache[row%3][j] = 3

}

// write back

// we chech "row-1>=0" because when row == 0, we have nothing to write back

if (row-1 >= 0)

{

for (int col = 0; col < n; ++col) cache[(row-1)%3][col] >>= 1;

write\_row(board, row-1, cache[(row-1)%3]);

}

if (row+2 < m)

{

cache[(row+2)%3] = read\_row(board, row+2);

}

}

// write back the last row

// because we write back row-1 after dealing with row

// but for the last row (i.e. m-1), we never reach to row == m

for (int col = 0; col < n; ++col) cache[(m-1)%3][col] >>= 1;

write\_row(board, row-1, cache[(m-1)%3]);

}

private int check(int[][] cache, int row, int col, int x, int y){

int ret = 0;

for(int i = Math.max(0, x - 1); i <= Math.min(row - 1, x + 1); i++){

for (int j = Math.max(0, y - 1); j <= Math.min(col - 1, y + 1); j++){

ret += cache[i%3][j] & 1;

}

}

ret -= cache[x%3][y] & 1;

return ret;

}

}

# LRU cache/NASA

class LRUCache {

HashMap<Pair<Integer,Integer>, Node> map;

Node head;

Node tail;

int size;

int cap;

class Node{

int x;

int y;

object img;

Node prev;

Node next;

public Node(int x, int y, object image){

this.x= x;

this.y = y;

img = image;

prev = null;

next = null;

}

}

public LRUCache(int capacity) {

map = new HashMap<>();

head = new Node(-1,-1, null);

tail = new Node(-1,-1, null);

head.next = tail;

tail.prev = head;

size = 0;

cap = capacity;

}

private void addToHead(Node n){

n.next = head.next;

n.next.prev = n;

head.next = n;

n.prev = head;

}

private void delete(Node n){

n.prev.next = n.next;

n.next.prev = n.prev;

}

public int get(int key) {

if (map.get(key) == null) return -1;

Node n = map.get(key);

delete(n);

addToHead(n);

return n.value;

}

public int get(int x, int y) {

Pair key = new Pair(x, y);

if (map.get(key) == null) return -1;

Node n = map.get(key);

delete(n);

addToHead(n);

return n.img;

}

public void put(int x, int y, object img) {

Pair key = new Pair(x, y);

if(map.get(key) != null){

Node n = map.get(key);

n.img = img;

delete(n);

addToHead(n);

}else{

if(size == cap){

map.remove(tail.prev.key);

delete(tail.prev);

size--;

}

Node n = new Node(x,y,img);

map.put(key,n);

addToHead(n);

size++;

}

}

Public void setLeastUpdated(object img){

Node n = tail.prev;

n.img = img;

delete(n);

addToHead(n);

}

}

/\*\*

\* Your LRUCache object will be instantiated and called as such:

\* LRUCache obj = new LRUCache(capacity);

\* int param\_1 = obj.get(key);

\* obj.put(key,value);

\*/

Hdfs: The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data.

# Hit counter

楼主本来以为会面到duplicated files,结果约到的是 log hit, leetcode 的一题Design Hit Counter。

要求用两种方法做，第一种，考虑到时间可以分为非常小，这个时候就用 queue + hashmap做，这样会浪费空间；

public class HitCounter {

// Stores the number of hits per timestamp

private Map<Integer, Integer> map;

// Stores the last 300 seconds timestamp records

private LinkedList<Integer> list;

/\*\* Initialize your data structure here. \*/

public HitCounter() {

map = new HashMap();

list = new LinkedList<Integer>();

}

/\*\* Record a hit.

@param timestamp - The current timestamp (in seconds granularity). \*/

public void hit(int timestamp) {

// If new hit, add to map and

// increment the hit count

if(!map.containsKey(timestamp)) {

map.put(timestamp, 1);

list.add(timestamp);

} else {

// If previously hit, just

// increment the hit count

// No need to add it to list

map.put(timestamp, map.get(timestamp) + 1);

}

// On each hit, do a repair

// so that we do not grow the

// list and map indefinitely

// untill a getHits is called.

repair(timestamp);

}

// This will ensure on any hit or getHit,

// we store data in the last 300 seconds

private void repair(int timestamp) {

while(!list.isEmpty()) {

if(list.getFirst() > timestamp - 300) {

break;

} else {

map.remove(list.removeFirst());

}

}

}

/\*\* Return the number of hits in the past 5 minutes.

@param timestamp - The current timestamp (in seconds granularity). \*/

public int getHits(int timestamp) {

repair(timestamp);

int ret = 0;

for(int i : list) {

ret += map.get(i);

}

return ret;

}

}

第二种，如果俭省空间，因为这个是一道很常见的题目，所以知道如何用circle array是基本知识。 你做完circle array之后，可以提一下新建一个Class， class 有两个属性，一个为 时间，另一个 为 hit次数。这样做也挺深空间的。

public class HitCounter {

int N;

int[] count;

int lastPosition;

int lastTime;

int sum;

/\*\* Initialize your data structure here. \*/

public HitCounter() {

N = 300;

count = new int[N];

lastPosition = 0;

lastTime = 0;

sum = 0;

}

/\*\* Record a hit.

@param timestamp - The current timestamp (in seconds granularity). \*/

public void hit(int timestamp) {

move(timestamp);

count[lastPosition]++;

sum++;

}

/\*\* Return the number of hits in the past 5 minutes.

@param timestamp - The current timestamp (in seconds granularity). \*/

public int getHits(int timestamp) {

move(timestamp);

return sum;

}

void move(int timestamp){

int gap = Math.min(timestamp-lastTime, N);

for(int i=0; i<gap;i++){

lastPosition = (lastPosition+1)%N;

sum -= count[lastPosition];

count[lastPosition] = 0;

}

lastTime = timestamp;

}

}

# Highest minimum sharpness

If(access.contains(folder\_name)) return true;

Node n = map.get(folder\_name);

Return hasAccess(n.parent.name);

}

Highest minimium sharpness

Brute force search

// n \* m board, a[n][m]

Int highest\_min = 0;

Void search(int row, int col, int current\_min)

{

If (current\_min <= highest\_min) return; //prune

If (col == m)

{

highest\_min = max(highest\_min, current\_min);

Return;

}

Current\_min = min(current\_min, a[row][col]);

For (int nxt\_row = max(0, row-1); nxt\_row <= min(n-1, row+1); ++nxt\_row)

{

Search(nxt\_row, col+1, current\_min);

}

}

Int main()

{

Highest\_min = 0;

For (int row = 0; row < n; ++row) search(row, 0, INT\_MAX);

Cout << highest\_min << endl;

} n \* 3^(m-1)

Transpose with square

Int[][] transpose

For()

Int [][] read(file fn, int x, int y, int xlen, int ylen)

Void write(file fn, int x, int y, int [][] board, int ylen, int xlen)

For (int row = 0; row < R; row+=size){

For int col = 0; col < C; col += sizw

{

Int xlen = min(size, R-row);

Int ylen = min(size, C- col);

Write(newfn,col, row, transpose(read(oldfn,row, col, xlen, ylen), ylen, xlen);

}

}

Dp[row][col] = at (row, col), what’s the highest minimum value.

[\\API](file:///\\API) to get a column in a board

Int[] getCol(int[][] board, int col, int row);

[\\API](file:///\\API) to update the board

Void update(int[][] board, int[] cache)

Int hms(int[][] board){

Int m = board.length();

Int n = board[0].length();

Int[] cache = new int[m][2];

For(int row = 0; row < n ; row++) cache[row][0] = getCol(board, 0, row);

Int res = -1;

For (int col = 1; col < n; col++){

For(int row = 0; row < n; row++){

Int row\_up = Math.max(0, row - 1);

Int row\_down = Math.min(m - 1, row);

cache[row][col] = getCol(board, col, row);

cache[row][col] = Math.min(Math.max(cache[row\_up][col - 1],Math.max(cache[row][col - 1],board[row\_down][col - 1])), cache[row][col]);

update(board, cache);

}

}

For(int row = 0; row < n ; row++) {

If (res < cache[row][n - 1]) res = cache[row][n - 1];

}

Return res;

}

# Has Access

大意就是给很多对(child, parent)这样的文件关系，再给定一个set，user对这个set中的文件夹有direct access，对这些文件夹的子文件夹也有access。  
现在给一个文件夹作为has\_access的参数，让你实现这个函数，查询user对它有没有访问权限。

/A  
|\_\_\_ /B   
    |\_\_\_ /C <-- access  
    |\_\_\_ /D  
|\_\_\_ /E <-- access  
    |\_\_\_ /F  
        |\_\_\_ /G  
  
  
folders = [  
  ('A', None),  
  ('B', 'A'),  
  ('C', 'B'),  
  ('D', 'B'),  
  ('E', 'A'),  
  ('F', 'E'),  
]  
  
  
access = set(['C', 'E'])  
  
has\_access(String folder\_name) -> boolean  
  
has\_access("B") -> false  
has\_access("C") -> true  
has\_access("F") -> true  
has\_access("G") -> true

Class Node{

String name;

Node parent;

Node child;

Node(String nm, Node parent){

name = nm;

This.parent = parent;

Child = null;

}

}

Node head;

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

LinkedList<String> generateLinkedLink(Set<Pair<String, String>> folders){

Head = new Node(null, null);

For(int I = 0; I < folder.size(); i++){

Node n = new Node(folders.get(i).getKey(), folders.get(i).getValue());

Map.put(folders.get(i).getKey(), n);

}

}

Boolean hasAccess(String folder\_name){

If(folder\_name == null) return false;

if (access.contains(folder\_name)) return true;

return hasAccess(map.get(folder\_name).parent.name);

}

# Combination sum

Int target\_sum;

Int a[0-(n-1)];

candidates = [2,3,6,7], target = 7,

search(0);

search(2), compos [2], i=1

search(5), compos [2 3 2] [2 2 3]

Set<vector<int>> ans;

vector<int> composition;

Void search(int cur\_sum, int pos) // cur\_sum = sum of elements in composition

{

If (cur\_sum == target\_sum)

{

Ans.insert(composition);

Return;

}

If (pos == n) return;

For (int I = pos; I < n; ++i)

If (cur\_sum + a[i] <= target\_sum)

{

Composition.push\_back(a[i]);

Search(cur\_sum+a[i], i);

Composition.pop\_back(a[i]); //backtrack

}

}

class Solution {

List<Integer> curList = new ArrayList<>();

List<List<Integer>> res = new ArrayList<>();

private void search(int[] candidates, int curSum, int pos, int target){

int n = candidates.length;

if (curSum == target){

List<Integer> copy = new ArrayList<>(curList);

// for (Integer e : curList){

// copy.add(e);

// }

res.add(copy);

return;

}

if (pos == n){

return;

}

for (int i = pos ; i < n; i++){

if (candidates[i] + curSum <= target){

curList.add(candidates[i]);

search(candidates,candidates[i] + curSum,i,target);

curList.remove(curList.size() - 1);

}

}

}

public List<List<Integer>> combinationSum(int[] candidates, int target) {

search(candidates,0, 0, target);

return res;

}

}