**Depth first search(dfs)**

**Time complexity of the Depth first search**

**Time complexity ,O (V+E) when implemented using the adjacency list.**

**Using recursion**

**bool visited[sz];**

**vector<int>adj[sz];**

**void dfs(int node)**

**{**

**visited[node]=true;**

**for (auto childnode:adj[node])**

**{**

**if (!visited[childnode])**

**{**

**dfs(childnode);**

**}**

**}**

**}**

**Using stack**

**bool visited[sz];**

**vector<int>adj[sz];**

**void dfs(int node)**

**{**

**stack<int>st;**

**st.push(node);**

**while(!st.empty())**

**{**

**int p=st.top();**

**st.pop();**

**vis[p]=1;**

**for (int i=0;i<adj[p].size();i++)**

**{**

**int ch=adj[p][i];**

**if (!vis[ch])**

**{**

**vis[ch]=1;**

**st.push(ch);**

**}**

**}**

**}**

**}**

**Breadth first search(bfs)**

**Time complexity of the Breadth first search**

**The time complexity of BFS is O(V + E), where V is the number of nodes and E is the number of edges .**

**int id[sz];**

**const int sz=1e5+1;**

**const int OO= 0x3f3f3f3f;**

**int dis[sz];**

**vector<int>adj[sz];**

**void bfs(int node)**

**{**

**dpp(dis,OO);**

**queue<int>q;**

**q.push(node);**

**dis[node]=1;**

**while(!q.empty())**

**{**

**int p=q.front();**

**q.pop();**

**for (int i=0;i<adj[p].size();i++)**

**{**

**int ch=adj[p][i];**

**if (dis[ch]==OO)**

**{**

**dis[ch]=dis[p]+1;**

**q.push(ch);**

**}**

**}**

**}**

**}**

**Multi source bfs**

BITMAP - Bitmap

*no tags*

There is given a rectangular bitmap of size *n\*m*. Each pixel of the bitmap is either white or black, but at least one is white. The pixel in *i*-th line and *j*-th column is called the pixel (*i,j*). The distance between two pixels **p1**=(*i1,j1*) and **p2**=(*i2,j2*) is defined as:

d(**p1**,**p2**)=|*i1-i2*|+|*j1-j2*|.

Task

Write a program which:

* reads the description of the bitmap from the standard input,
* for each pixel, computes the distance to the nearest white pixel,
* writes the results to the standard output.

Input

The number of test cases t is in the first line of input, then t test cases follow separated by an empty line. In the first line of each test case there is a pair of integer numbers*n, m* separated by a single space, *1<=n <=182*, *1<=m<=182*. In each of the following *n* lines of the test case exactly one zero-one word of length *m,* the description of one line of the bitmap, is written. On the *j*-th position in the line (*i+1*), *1 <= i <= n*, *1 <= j <= m*, is '1' if, and only if the pixel (*i,j*) is white.

Output

In the *i*-th line for each test case, *1<=i<=n*, there should be written *m* integers f(*i,1*),...,f(*i,m*) separated by single spaces, where f(*i,j*) is the distance from the pixel (*i,j*) to the nearest white pixel.

**int M=1e9+7;**

**const int sz=2e2+1;**

**const int OO= 0x3f3f3f3f;**

**int arr[sz][sz];**

**int dx[4]={0,0,1,-1};**

**int dy[4]={1,-1,0,0};**

**bool valid(int x,int y,int n,int m,int dist)**

**{**

**return (x>=0&&x<n&&y>=0&&y<m&&dist<arr[x][y]);**

**}**

**void bfs(int x,int y,int n,int m)**

**{**

**arr[x][y]=0;**

**queue<int>q;**

**q.push(x);**

**q.push(y);**

**arr[x][y]=0;**

**while(!q.empty())**

**{**

**int px=q.front();**

**q.pop();**

**int py=q.front();**

**q.pop();**

**int dist=arr[px][py]+1;**

**for (int i=0;i<4;i++)**

**{**

**int curx=px+dx[i];**

**int cury=py+dy[i];**

**if (valid(curx,cury,n,m,dist))**

**{**

**arr[curx][cury]=dist;**

**q.push(curx);**

**q.push(cury);**

**}**

**}**

**}**

**}**

**int main()**

**{**

**//myf.open("file.txt");**

**//freopen("task.in", "r", stdin);**

**//freopen("output.txt", "w", stdout);**

**ios\_base::sync\_with\_stdio(0),cin.tie(0),cout.tie(0);**

**int t;**

**cin>>t;**

**while(t--)**

**{**

**int n,m;**

**cin>>n>>m;**

**vector<pii>st\_nodes;**

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

**{**

**for (int j=0;j<m;j++)**

**{**

**char x;**

**cin>>x;**

**if (x=='1')**

**{**

**arr[i][j]=-OO;**

**st\_nodes.push\_back({i,j});**

**}**

**else {**

**arr[i][j]=OO;**

**}**

**}**

**}**

**for (int i=0;i<st\_nodes.size();i++)**

**{**

**bfs(st\_nodes[i].F,st\_nodes[i].S,n,m);**

**}**

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

**{**

**for (int j=0;j<m;j++)**

**{**

**cout<<arr[i][j]<<" ";**

**}**

**cout<<endl;**

**}**

**cout<<endl;**

**}**

**return 0;**

**}**

**Disjoint set**

**Time complexity of the disjoint set**

**O(Mα(N))**

**When you use the weighted-union operation with path compression it takes log \* N for each union find operation, where N is the number of elements in the set.**

**log \*N is the iterative function that computes the number of times you have to take the log of N before the value of N reaches 1.**

**α(N): is a slowly growing function <=5 which make the over all complexity O(5\*M)**

**where m is the number of operations n is the number of elements**

**int id[sz];**

**void disjoint\_intialize()**

**{**

**for (int i=0;i<sz;i++)**

**{**

**id[i]=i;**

**}**

**}**

**int root(int x)**

**{**

**while(id[x]!=x)**

**{**

**id[x]=id[id[x]];**

**x=id[x];**

**}**

**return x;**

**}**

**void disjoint\_union(int x,int y)**

**{**

**int p=root(x);**

**int q=root(y);**

**id[p]=q;**

**}**

**bool disjoint\_find(int x,int y) // if both has the same root they are connected and there's cycle here**

**{**

**return (root(x)==root(y));**

**}**

**Kruskal’s Algorithm (minimum spanning tree)**

**Time complexity of the Kruskal’s Algorithm**

**In Kruskal’s algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(E log V), which is the overall Time Complexity of the algorithm .**

**int id[sz];**

**void disjoint\_intialize()**

**{**

**for (int i=0;i<sz;i++)**

**{**

**id[i]=i;**

**}**

**}**

**int root(int x)**

**{**

**while(id[x]!=x)**

**{**

**id[x]=id[id[x]];**

**x=id[x];**

**}**

**return x;**

**}**

**void disjoint\_union(int x,int y)**

**{**

**int p=root(x);**

**int q=root(y);**

**id[p]=q;**

**}**

**bool disjoint\_find(int x,int y)**

**{**

**return (root(x)==root(y));**

**// if both has the same root they are connected and there's cycle here**

**}**

**int kurksal\_algorithm(vector<pair<int,pii>>&adj)**

**{**

**sort(adj.begin(),adj.end());**

**ll minicost=0;**

**for (int i=0;i<adj.size();i++)**

**{**

**int x=adj[i].S.F;**

**int y=adj[i].S.S;**

**int cost=adj[i].F;**

**if (!disjoint\_find(x,y))**

**{**

**disjoint\_union(x,y);**

**minicost+=cost;**

**}**

**}**

**return minicost;**

**}**

**int main()**

**{**

**//myf.open("file.txt");**

**//freopen("task.in", "r", stdin);**

**//freopen("output.txt", "w", stdout);**

**ios\_base::sync\_with\_stdio(0),cin.tie(0),cout.tie(0);**

**int n,m;**

**cin>>n>>m;**

**vector<pair<int,pair<int,int>>>adj;**

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

**{**

**ll x,y,w;**

**cin>>x>>y>>w;**

**adj.push\_back({w,{x,y}});**

**}**

**disjoint\_intialize();**

**cout<<kurksal\_algorithm(adj)<<endl;**

**return 0;**

**}**

****