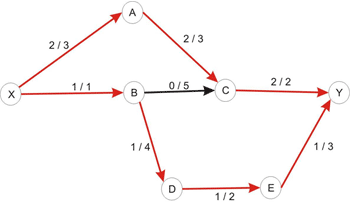
# CSE Project 200 - Network Flow

In graph theory, a **flow network** (also known as a **transportation network**) is a directed graph where each edge has a **capacity** and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge. Often in Operations Research, a directed graph is called a **network**, the vertices are called **nodes** and the edges are called **arcs**. A flow must satisfy the restriction that the amount of flow into a node equals the amount of flow out of it, except when it is a **source**, which has more outgoing flow, or **sink**, which has more incoming flow. A network can be used to model traffic in a road system, fluids in pipes, currents in an electrical circuit, or anything similar in which something travels through a network of nodes.

When the capacities are integers, the runtime of Ford-Fulkerson is bounded by O(Ef), where E is the number of edges in the graph and f is the maximum flow in the graph. This is because each augmenting path can be found in O(E) time and increases the flow by an integer amount which is at least 1.

A variation of the Ford–Fulkerson algorithm with guaranteed termination and a runtime independent of the maximum flow value is the Edmonds–Karp algorithm, which runs in O(VE2) time.

A typical network flow would be like this: "A list of pipes is given, with different flow-capacities. These pipes are connected at their endpoints. What is the maximum amount of water that you can route from a given starting point to a given ending point?" or equivalently "A company owns a factory located in city X where products are manufactured that need to be transported to the distribution center in city Y. You are given the one-way roads that connect pairs of cities in the country, and the maximum number of trucks that can drive along each road. What is the maximum number of trucks that the company can send to the distribution center?"



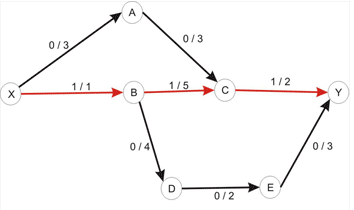
To solve a network flow problem we first have to understand two basic concepts :

* Residual Networks
* Augmenting Paths

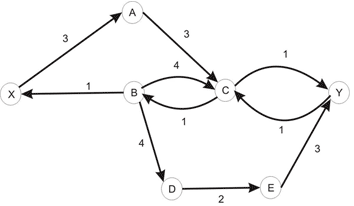
Residual Networks:

The residual network has the same vertices as the original network, and one or two edges for each edge in the original. More specifically, if the flow along the edge x-y is less than the capacity there is a forward edge x-y with a capacity equal to the difference between the capacity and the flow (this is called the residual capacity), and if the flow is positive there is a backward edge y-x with a capacity equal to the flow on x-y.

Suppose this is a graph with capacities attached to each edge.



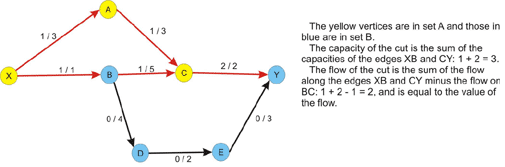
A residual network of this graph will be like this ->



Augmenting Path:

An augmenting path is simply a path from the source to the sink in the residual network, whose purpose is to increase the flow in the original one. It is important to understand that the edges in this path can point the "wrong way" according to the original network. The path capacity of a path is the minimum capacity of an edge along that path.

A cut in a flow network is simply a partition of the vertices in two sets, let's call them A and B, in such a way that the source vertex is in A and the sink is in B. The capacity of a cut is the sum of the capacities of the edges that go from a vertex in A to a vertex in B. The flow of the cut is the difference of the flows that go from A to B (the sum of the flows along the edges that have the starting point in A and the ending point in B), respectively from B to A, which is exactly the value of the flow in the network, due to the entering flow equals leaving flow - property, which is true for every vertex other than the source and the sink.



Notice that the flow of the cut is less or equal to the capacity of the cut due to the constraint of the flow being less or equal to the capacity of every edge. This implies that the maximum flow is less or equal to every cut of the network. This is where the max-flow min-cut theorem comes in and states that the value of the maximum flow through the network is exactly the value of the minimum cut of the network.

The value of the flow must equal the value of the cut, and since every flow is less or equal to every cut, this must be a maximum flow, and the cut is a minimum cut as well.

Augmenting-Path Algorithms

The neat part of the Ford-Fulkerson algorithm described above is that it gets the correct result no matter how we solve (correctly!!) the sub-problem of finding an augmenting path. However, every new path may increase the flow by only 1, hence the number of iterations of the algorithm could be very large if we carelessly choose the augmenting path algorithm to use. The function *max\_flow* will look like this, regardless of the actual method we use for finding augmenting paths:

int max\_flow()

result = 0

while (true)

// the function find\_path returns the path capacity of the

augmenting path found

path\_capacity = find\_path()

// no augmenting path found

if (d = 0) exit while

else result += path\_capacity

end while

return result

We will normally use a BFS (Breadth-First-Search) to find an augmenting path when solving basic maximum flow problems. Below are some applications:

**Problem 1**

**LightOJ- 1153 (Internet Bandwidth)**

# Problem Statement

On the Internet, machines (nodes) are richly interconnected, and many paths may exist between a given pair of nodes. The total message-carrying capacity (bandwidth) between two given nodes is the maximal amount of data per unit time that can be transmitted from one node to the other. Using a technique called packet switching; this data can be transmitted along several paths at the same time.

You must write a program that computes the bandwidth between two given nodes in a network, given the individual bandwidths of all the connections in the network. In this problem, assume that the bandwidth of a connection is always the same in both directions (which is not necessarily true in the real world).

# Input

Input starts with an integer **T (≤ 30)**, denoting the number of test cases.

Every description starts with a line containing an integer **n (2 ≤ n ≤ 100)**, which is the number of nodes in the network. The nodes are numbered from **1** to **n**. The next line contains three numbers **s**, **t**, and **c**. The numbers **s** and **t** are the source and destination nodes, and the number **c (c ≤ 5000, s ≠ t)** is the total number of connections in the network. Following this are **c** lines describing the connections. Each of these lines contains three integers: the first two are the numbers of the connected nodes, and the third number is the bandwidth of the connection. The bandwidth is a non-negative number not greater than **1000**.

There might be more than one connection between a pair of nodes, but a node cannot be connected to itself. All connections are bi-directional, i.e. data can be transmitted in both directions along a connection, but the sum of the amount of data transmitted in both directions must be less than the bandwidth.

# Output

For each case of input, print the case number and the total bandwidth between the source node **s** and the destination node **t**.

**Solution:**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<iostream>

#include<string>

#include<sstream>

#include<ctype.h>

#include<vector>

#include<map>

#include<queue>

#include<algorithm>

#include<set>

#define pb push\_back

#define SZ(a) (int)a.size()

#define csprnt printf("Case %d: ", cas++);

#define MAX 100010

#define INF (1<<30)

using namespace std;

int source, sink, par[110], dist[110][110], flow; vector<int>adj[110];

bool aug\_path()

{

int i, u, v;

queue<int>Q;

Q.push(source);

memset(par, -1, sizeof par);

while(!Q.empty())

{

u = Q.front(); Q.pop();

for(i=0;i<SZ(adj[u]);i++)

{

v = adj[u][i];

if(par[v]==-1 && dist[u][v]>0)

{

par[v]=u;

Q.push(v);

if(v==sink)

return true;

}

}

}

return false;

}

void path\_upd(int v)

{

int u = par[v];

flow=min(flow, dist[u][v]);

if(u!=source) path\_upd(u);

dist[u][v]-=flow;

dist[v][u]+=flow;

}

int maxflow()

{

int ret=0;

while(aug\_path())

{

flow=INF;

path\_upd(sink);

ret+=flow;

}

return ret;

}

int main()

{

int t, cas=1;

scanf("%d", &t);

while(t--)

{

int n, i, j, c, u, v, w;

memset(dist, 0, sizeof dist);

scanf("%d", &n);

for(i=0;i<=n;i++) adj[i].clear();

scanf("%d%d%d", &source, &sink, &c);

for(i=0;i<c;i++)

{

scanf("%d%d%d", &u, &v, &w);

if(dist[u][v]==0)

{

adj[u].pb(v);

adj[v].pb(u);

}

dist[u][v]+=w;

dist[v][u]+=w;

}

int ans = maxflow();

csprnt; printf("%d\n", ans);

}

return 0;

}

**Problem 2**

**LightOJ- 1155 (Power Transmission)**

DESA is taking a new project to transfer power. Power is generated by the newly established plant in Barisal. The main aim of this project is to transfer Power in Dhaka. As Dhaka is a megacity with almost 10 million people DESA wants to transfer maximum amount of power through the network. But as always occurs in case of power transmission it is tough to resist loss. So they want to use some regulators whose main aims are to divert power through several outlets without any loss.

Each such regulator has different capacity. It means if a regulator gets 100 units of power and its capacity is 80 units then remaining 20 units of power will be lost. Moreover each unidirectional link (connectors among regulators) has a certain capacity. A link with capacity 20 units cannot transfer power more than 20 units. Each regulator can distribute the input power among the outgoing links so that no link capacity is over flown. DESA wants to know the maximum amount of power which can be transmitted throughout the network so that no power loss occurs. That is the job you have to do.

# Input

Input starts with an integer **T (≤ 50)**, denoting the number of test cases.

The input will start with a positive integer **N (1 ≤ N ≤ 100)** indicates the number of regulators. The next line contains **N** positive integers indicating the capacity of each regulator from **1** to **N**. All the given capacities will be positive and not greater than **1000**. The next line contains another positive integer **M** which is the number of links available among the regulators. Each of the following **M** lines contains three positive integers **i j C**. **'i'** and **'j'** are the regulator index **(1 ≤ i, j ≤ N, i ≠ j, 1 ≤ C ≤ 1000)** and **C** is the capacity of the link. Power can be transferred from **ith** regulator to **jth** regulator. From a regulator **i** to another regulator **j**, there can be at most one link.

The next line contains two positive integers **B** and **D (1 ≤ B, D and B + D ≤ N)**. **B** is the number of regulators which are the entry point of the network. Power generated in Barisal must enter in the network through these entry points. Similarly **D** is the number of regulators connected to Dhaka. These links are special and have infinite capacity. Next line will contain **B+D** integers each of which is an index of regulator. The first **B** integers are the index of regulators connected with Barisal. Regulators connected with Barisal are not connected with Dhaka.

# Output

For each case of input, print the case number and the maximum amount of power which can be transferred from Barisal to Dhaka.

# Analysis

In this problem, every edge as well as every node have a limited amount of capacity. For solving this type of problem we can use a smart trick which is known as “Node Splitting”. By using this technique we split a node in two and between these two nodes we create an edge with the same capacity of the nodes capacity. Thus we can give a node a certain amount of capacity.

# Solution

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<iostream>

#include<string>

#include<sstream>

#include<ctype.h>

#include<vector>

#include<map>

#include<queue>

#include<math.h>

#include<algorithm>

#include<set>

#define pb push\_back

#define PI acos(-1.0)

#define SZ(a) (int)a.size()

#define csprnt printf("Case %d: ", cas++);

#define EPS 1e-9

#define MAX 100010

#define INF (1<<30)

#define pii pair<int, int>

using namespace std;

int source, sink, par[210], dist[210][210], flow;

vector<int>adj[210];

bool aug\_path()

{

int i, u, v;

queue<int>Q;

Q.push(source);

memset(par, -1, sizeof par);

while(!Q.empty())

{

u = Q.front(); Q.pop();

for(i=0;i<SZ(adj[u]);i++)

{

v = adj[u][i];

if(par[v]==-1 && dist[u][v]>0)

{

par[v]=u;

Q.push(v);

if(v==sink)

return true;

}

}

}

return false;

}

void path\_upd(int v)

{

int u = par[v];

flow=min(flow, dist[u][v]);

if(u!=source) path\_upd(u);

dist[u][v]-=flow;

dist[v][u]+=flow;

return;

}

int maxflow()

{

int ret=0;

while(aug\_path())

{

flow=INF;

path\_upd(sink);

ret+=flow;

}

return ret;

}

int main()

{

int t, cas=1;

scanf("%d", &t);

while(t--)

{

int n, i, j, c, u, v, w;

memset(dist, 0, sizeof dist);

scanf("%d", &n);

for(i=0;i<=201;i++) adj[i].clear();

for(i=1;i<=n;i++)

{

scanf("%d", &w);

adj[i].pb(i+100);

dist[i][i+100]=w;

}

source=0, sink=201;

scanf("%d", &c);

for(i=0;i<c;i++)

{

scanf("%d%d%d", &u, &v, &w);

if(dist[u+100][v]==0)

{

adj[u+100].pb(v);

adj[v].pb(u+100);

}

dist[u+100][v]+=w;

}

int b, d;

scanf("%d%d", &b, &d);

for(i=0;i<b;i++)

{

scanf("%d", &u);

adj[source].pb(u);

dist[source][u]=INF;

}

for(i=0;i<d;i++)

{

scanf("%d", &u);

adj[u+100].pb(sink);

dist[u+100][sink]=INF;

}

int ans = maxflow();

csprnt; printf("%d\n", ans);

}

return 0;

}

***Problem 3***

***LightOJ- 1149 (Factors and Multiples)***

You will be given two sets of integers. Let's call them set **A** and set **B**. Set **A** contains **n** elements and set **B** contains **m** elements. You have to remove **k1** elements from set **A** and **k2** elements from set **B** so that of the remaining values no integer in set **B** is a multiple of any integer in set **A**. **k1** should be in the range **[0, n]** and **k2** in the range **[0, m]**.

You have to find the value of **(k1 + k2)** such that **(k1 + k2)** is as low as possible. **P** is a multiple of **Q** if there is some integer **K** such that **P** = **K \* Q**.

Suppose set **A** is **{2, 3, 4, 5}** and set **B** is **{6, 7, 8, 9}**. By removing **2** and **3** from **A** and **8** from B, we get the sets **{4, 5}** and **{6, 7, 9}**. Here none of the integers **6, 7** or **9** is a multiple of **4** or **5**.

So for this case the answer is **3** (**two** from set **A** and **one** from set **B**).

# Input

Input starts with an integer **T (≤ 50)**, denoting the number of test cases.

The first line of each case starts with an integer **n** followed by **n** positive integers. The second line starts with **m** followed by **m** positive integers. Both **n** and **m** will be in the range **[1, 100]**. Each element of the two sets will fit in a **32** bit signed integer.

# Output

For each case of input, print the case number and the result.

# Analysis

Here we have to set every capacity of the graphs edges as “1” and the total set of nodes can be divided into two groups. This type of flow problems are known as “Bipartite Matching”. This problem can be implemented very smartly using a dfs, which is very quick and easy to code.

**Solution:**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<iostream>

#include<string>

#include<sstream>

#include<ctype.h>

#include<vector>

#include<map>

#include<queue>

#include<math.h>

#include<algorithm>

#include<set>

#define pb push\_back

#define PI acos(-1.0)

#define SZ(a) (int)a.size()

#define csprnt printf("Case %d: ", cas++);

#define MAX 100010

#define INF (1<<30)

#define pii pair<int, int>

using namespace std;

vector<int>adj[105]; int a[105], b[105], n, m, arr[205]; bool col[205];

bool dfs(int now)

{

int i, j, end;

if(col[now]) return false;

col[now]=true;

for(i=0;i<SZ(adj[now]);i++)

{

end = adj[now][i];

if((arr[end]==-1) || (dfs(arr[end])))

{

arr[end] = now;

return true;

}

}

return false;

}

int match()

{

memset(arr, -1, sizeof arr);

int i, j, cnt=0;

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

{

memset(col, false, sizeof col);

if(dfs(i))

cnt++;

}

return cnt;

}

int main()

{

int t, cas=1;

scanf("%d", &t);

while(t--)

{

int i, j;

scanf("%d", &n);

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

{

adj[i].clear();

scanf("%d", &a[i]);

}

scanf("%d", &m);

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

scanf("%d", &b[i]);

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

{

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

{

if((b[j]%a[i])==0)

adj[i].pb(j);

}

}

int sol = match();

csprnt;

printf("%d\n", sol);

}

return 0;

}

***Problem 4***

***LightOJ- 1404 (Sending Secret Messages)***

Alice wants to send Bob some confidential messages. But their internet connection is not secured enough. As their names have been used in many networking schemes, they are very rich now. So, they don't want to send encoded messages, they want to use secured dedicated connection for them. So, they talked to some ISPS (Internet Service Providers) about their problem. Only they get is that there are **N** routers in the network, some of them share bidirectional links. Each link has a capacity, and for each KB of data passing through this link, they have to pay some money. Assume that Alice is connected with the **1st** router and Bob is connected to the **Nth** router.

Now Alice wants to send **P** KB of data to Bob. You have to find the minimum amount of money they have to pay to achieve their goal.

# Input

Input starts with an integer **T (≤ 50)**, denoting the number of test cases.

Each case starts with a blank line. Next line contains three integers **N (2 ≤ N ≤ 50)**, **M (0 ≤ M ≤ N\*(N-1)/2)** and **P (1 ≤ P ≤ 1000)**, where **M** denotes the number of bidirectional links. Each of the next **M** lines contains four integers **u v w c (1 ≤ u, v ≤ N, u ≠ v, 1 ≤ w, c ≤ 100)**, meaning that there is a link between router **u** and **v**, and at most **c KB** data can be sent through this link, and each KB of data through this link will cost **w**. You can assume that there will be at most one connection between a pair of routers.

# Output

For each case, print the case number and the minimum amount of money required or **"impossible"** if it's not possible to send **P** KB of data.

# Analysis

In this type of problems, the edges have a cost as well as a capacity. These type of maximum flow problems are known as “Min-Cost-Max-Flow” (Minimum Cost Maximum Flow). This type of problem is not very different from the traditional flow problems, and only changing the augmenting path function, we can solve these problems. Previously we used either BFS or DFS to solve the problems, this time we have to use a Shortest Path algorithm as the Augmenting Path function to ensure that the current path chosen has the minimum cost among the currently available paths. In the below implemented solution we have used “Dijkstra’s Algorithm for SSSP” as the Augmenting Path function.

**Solution**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<iostream>

#include<string>

#include<sstream>

#include<ctype.h>

#include<vector>

#include<map>

#include<queue>

#include<math.h>

#include<algorithm>

#include<set>

#define pb push\_back

#define PI acos(-1.0)

#define SZ(a) (int)a.size()

#define csprnt printf("Case %d: ", cas++);

#define EPS 1e-9

#define MAX 100010

#define INF (1<<30)

#define pii pair<int, int>

using namespace std;

int cst[105][105], par[105], source, sink, flow, dist[105];

struct pq{

int cost, node;

pq(int x, int y) { node=x, cost=y;}

bool operator<(const pq &b)const

{return (cost>b.cost);}};

vector<pq>adj[105];

void addedge(int u, int v, int c, int w)

{

adj[u].pb(pq(v, w)); adj[v].pb(pq(u, -w));

cst[u][v] = c, cst[v][u] = 0;

}

bool aug\_path()

{

int i, now, nxt, nc, xc;

priority\_queue<pq>Q;

Q.push(pq(source, 0));

for(i=0;i<104;i++) dist[i]=INF;

memset(par, -1, sizeof par);

dist[source] = 0;

while(!Q.empty())

{

now = Q.top().node, nc = Q.top().cost; Q.pop();

if(now==sink) return true;

for(i=0;i<SZ(adj[now]);i++)

{

nxt = adj[now][i].node, xc = adj[now][i].cost;

if((dist[nxt] > (xc+dist[now])) && (cst[now][nxt]>0))

{

dist[nxt] = dist[now]+xc;

par[nxt] = now;

Q.push(pq(nxt, dist[now]+xc));

}

}

}

if(dist[sink]!=INF) return true;

return false;

}

void path\_upd(int v)

{

int u = par[v];

flow = min(flow, cst[u][v]);

if(u!=source) path\_upd(u);

cst[u][v]-=flow;

cst[v][u]+=flow;

return;

}

int maxflow(int p)

{

int ret=0;

while(p>0)

{

if(!(aug\_path())) break;

flow=INF;

path\_upd(sink);

ret+=(min(flow, p)\*dist[sink]);

p-=flow;

}

if(p>0) return -1;

return ret;

}

int main()

{

int t, cas=1;

scanf("%d", &t);

while(t--)

{

int i, j, k, n, m, p, u, v, c, w;

scanf("%d%d%d", &n, &m, &p);

for(i=0;i<104;i++) adj[i].clear();

memset(cst, 0, sizeof cst);

for(i=1;i<=n;i++)

addedge(i, i+n, INF, 0);

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

{

scanf("%d%d%d%d", &u, &v, &c, &w);

addedge(u+n, v, c, w);

addedge(v+n, u, c, w);

}

source=1, sink=n;

int ans = maxflow(p);

csprnt;

if(ans==-1)

printf("impossible\n");

else

printf("%d\n", ans);

}

return 0;

}

***Problem 5***

***LightOJ- 1405 (The Great Escape)***

Walter White finally discovered the medicine for a dangerous disease. That's why he is in danger, because some people are targeting him to get the formula. So, now he and his co-workers have to leave the city.

Assume that the city can be modeled as an **M x N** grid, where a cell containing a **'\*'** means that a co-worker lives in that place, a cell containing a **'.'** means it's empty.

So, Walter informed all of them by phone to leave the city at once and he has laid down a rule that no one will cross path in their way out of the city. Since if two or more persons meet at same cell (even in the boundary cells), there is a big possibility that they all might get caught. Walter doesn't want that, so he needs to know whether all of them can get out of the city. So, he asked you to help him.

For simplicity, assume that if one person reaches any of the border cells of the grid, he is considered to be out of the city. And from any cell, a person can move only to its four adjacent cells (north, south, east or west).

# Input

Input starts with an integer **T (≤ 20)**, denoting the number of test cases.

Each case starts with a line containing two integers **M** and **N (1 ≤ M, N ≤ 100)**. Each of the next **M** lines contains **N** characters denoting the map. Each characters is either **'.'** or **'\*'**. Total number of persons in a map will be at most **2\*(M+N)**.

# Output

For each case, print the case number and **'yes'** if it's possible for them to get out of the city maintaining the restrictions. Otherwise print **'no'**.

# Analysis

This is a pretty straight forward flow problem, we can visualize each cell on the grid as a node and the sides of each cell as edges. The difficulty with this problem then comes down to the number of nodes.

For this problem, the number of nodes (after node splitting) is 2\*100\*100 or 20,000 and the number of edges is roughly 4\*100\*100 or 40,000. This will definitely run out of time if we implemented normal max-flow algorithm. For solving this type of problem we need to implement “Dinic’s Algorithm”, which when choosing the augmented path, chooses a number of paths at once rather than one path at a time. This cuts the complexity down to O(V2E) rather than the normal O(VE2).

**Solution**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<iostream>

#include<string>

#include<sstream>

#include<ctype.h>

#include<vector>

#include<map>

#include<queue>

#include<math.h>

#include<algorithm>

#include<set>

#define pb push\_back

#define SZ(a) (int)a.size()

#define csprnt printf("Case %d: ", cas++);

#define MAX 100010

#define ll long long

#define INF (1<<30)

#define pii pair<int, int>

#define MP make\_pair

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

using namespace std;

struct node{

int u, v, cap, revind;

node(int c=0, int a=0, int b=0, int d=0)

{

u=c, v=a, cap=b, revind=d;

}

}; node edge[400000];

vector<int>adj[20010]; vector<pii>ed;

int source, sink, pind[20010], flow, edge\_no; char board[105][105];

void addedge(int u, int v, int cap)

{

edge[edge\_no] = node(u, v, 1, edge\_no+1);

adj[u].pb(edge\_no);

edge[edge\_no+1] = node(v, u, 0, edge\_no);

adj[v].pb(edge\_no+1);

edge\_no+=2;

}

bool aug\_path()

{

int i, u, used=0; queue<int>Q;

Q.push(source);

memset(pind, -1, sizeof pind);

pind[source] = -2;

while(!Q.empty())

{

u = Q.front(), Q.pop();

for(i=0;i<SZ(adj[u]);i++)

{

node now = edge[adj[u][i]];

if(pind[now.v]!=-1) continue;

if(now.cap<=0) continue;

pind[now.v] = adj[u][i];;

if(now.v!=sink) Q.push(now.v);

}

}

return (pind[sink]!=-1);

}

void path\_upd(int v)

{

node now = edge[pind[v]];

flow = min(flow, now.cap);

if(now.u!=source) path\_upd(now.u);

edge[pind[v]].cap -= flow;

edge[now.revind].cap += flow;

}

bool maxflow(int cnt)

{

int i, u, v, j, ret=0;

while(aug\_path())

{

for(i=0;i<SZ(ed);i++)

{

u = ed[i].first, j = ed[i].second;

if(pind[u]==-1) continue;

if(edge[j].cap<=0) continue;

pind[sink] = j;

flow=INF;

path\_upd(sink);

ret+=flow;

if(ret==cnt) return true;

}

}

return false;

}

int main()

{

int t, cas=1;

scanf("%d", &t);

while(t--)

{

edge\_no=0;

int N, M, i, j, k, nodes, nx, ny, u, v;

int req=0;

scanf("%d%d ", &N, &M);

for(i=0;i<N;i++) scanf("%s", board[i]);

nodes = N\*M; ed.clear();

source=0, sink = 2\*nodes + 1;

for(i=0;i<=sink;i++) adj[i].clear();

for(i=0;i<N;i++)

{

for(j=0;j<M;j++)

{

u = (i\*M)+j+1;

addedge(u, u+nodes, 1);

if(board[i][j]=='\*')

{

addedge(source, u, 1);

req++;

}

if(i==0 || i==N-1 || j==0 || j==M-1)

{

addedge(u+nodes, sink, 1);

ed.pb(MP(u+nodes, edge\_no-2));

}

for(k=0;k<4;k++)

{

nx = i+dx[k], ny = j+dy[k];

if(nx<0 || nx==N || ny<0 || ny==M) continue;

v = (nx\*M)+ny+1;

addedge(u+nodes, v, 1);

}

}

}

int sol = maxflow(req); csprnt;

if(sol) printf("yes\n");

else printf("no\n");

}

return 0;

}