Bài 1: DIGOKEYS - Find the Treasure (medium)

https://www.spoj.com/problems/DIGOKEYS/

Có N hộp đã bị khóa được đánh số từ 1 đến N trừ hộp thứ nhất. Hộp thứ N chứa kho báu, N-1 hộp còn lại chứa m chìa khóa để mở m hộp có vị trí tương ứng. Mỗi lần mở chỉ được mở 1 hộp. Bắt đầu với hộp số 1 không bị khóa. Tìm cách để có thể mở được hộp N với số hộp cần mở là nhỏ nhất. Nếu không có cách nào mở được hộp, in ra -1.

Input:

Dòng đầu tiên T là tổng số test case.

Ứng với mỗi test case sẽ có input như sau:

Dòng thứ nhất là số N là tổng số hộp.

N-1 dòng tiếp theo, ở dòng thứ *ith* có số M là tổng số chìa khóa bên trong hộp này. M số tiếp theo của dòng này là Mj tương ứng là chìa khóa có thể mở được hộp thứ *jth*.

Output:

Với mỗi test case in ra số q là số hộp cần mở.

Dòng tiếp theo bao gồm q số tương ứng là các hộp cần mở. Nếu có nhiều cách để mở, in ra cách có thứ tự mở hộp theo từ điển là nhỏ nhất.

Điều kiện:

1 <= T <= 10

2 <= N <= 100000

1 <= M <= 10

Ví dụ:

1

5

1 4

1 5

1 5

2 2 3

Ví dụ này có 1 test case. Tổng số có 3 hộp. Hộp 1 chứa 1 chìa khóa có thể mở được hộp 4. Hộp 2 chứa 1 chìa khóa mở được hộp 5. Hộp 3 chứa 1 chìa khóa mở được hộp 5. Hộp 4 chứa 2 chìa khóa mở được hộp 2 và 3. Có 2 cách mở hộp 5 như sau:

Cách 1: Mở hộp 1 -> mở hộp 4 -> mở hộp 2 -> mở hộp 5. Các hộp cần mở để tới hộp 5 {1, 4, 2}

Cách 2: Mở hộp 1 -> mở hộp 4 -> mở hộp 3 -> mở hộp 5. Các hộp cần mở để tới hộp 5 {1, 4, 3}

Cả 2 cách đều có thể mở hộp 5 sau 3 lần mở hộp nhưng cách 1 là đáp án đúng vì {1, 4, 2} được xem là nhỏ hơn khi sắp xếp các cách theo thứ tự từ điển.

Brute-force:

Tìm tất cả các cách có thể để mở được hộp N.

Nếu có nhiều cách, dùng hàm để kiểm tra xem cách nào có thứ tự nhỏ hơn.

Nếu không tìm được cách nào, in ra -1

Time complexity: O(M^N)

Code:

```
#include <bits/stdc++.h>

using namespace std;


vector<vector<int>> adj;

vector<int> ans;

int n;

// Hàm kiểm tra xem số hộp cần mở trong vector a có nhỏ hơn cách trong vector b hay không

bool checkSmaller(vector<int> a, vector<int> b) {

  // Nếu có cùng số phần tử, kiểm tra từng phần tử 1. Nếu có phần tử của vector a nhỏ hơn phần tử

  // của vector b, trả về true

  // Nếu 2 vector có size khác nhau, kiểm tra xem size của a có nhỏ hơn size của b không

  if (a.size() == b.size()) {

    for (int i = 0; i < a.size(); ++i) {

      if (a[i] < b[i])

        return true;

    }

    return false;

  } else

    return a.size() < b.size();
```

```
}

// duyệt xem từ hộp u mở được những hộp nào
// nếu mở tới hộp cuối cùng, lưu lại cách mở nhỏ nhất.
// vector used dùng để đánh dấu hộp được mở rồi
// vector path dùng để lưu cách mở
void tryOpen(int u, vector<bool> used, vector<int> path) {
    used[u] = true;
    if (u == n - 1) {
        if (ans.empty() || checkSmaller(path, ans)) {
            ans = path;
        }
        return;
    }
    for (auto nextBox : adj[u]) {
        if (!used[nextBox]) {
            path.push_back(nextBox);
            tryOpen(nextBox, used, path);
            path.pop_back();
        }
    }
}

int main() {
    int T;
    cin >> T;
    while (T--) {
        cin >> n;
        adj = vector<vector<int>>(n);
```

```
    ans.clear();

    for (int i = 0; i < n - 1; ++i) {
      int m;
      cin >> m;
      for (int j = 0; j < m; ++j) {
        int nextBox;
        cin >> nextBox;
        // dùng theo index-0 nên giảm chỉ số của hộp đi 1 trước khi lưu vào list
        --nextBox;
        adj[i].push_back(nextBox);
      }
    }
    vector<bool> used(n);
    vector<int> path;
    path.push_back(0);
    tryOpen(0, used, path);
    if (ans.empty())
      cout << -1 << '\n';
    else {
      cout << ans.size() - 1 << '\n';
      for (int i = 0; i < ans.size() - 1; ++i)
        cout << ans[i] + 1 << ' ';
      cout << '\n';
    }
  }
}
```

Improvement by using breadth-first search

Dùng breadth-first search để tìm cách mở ngắn nhất từ hộp 1 tới hộp N

Để tìm được cách có các hộp cần mở là nhỏ nhất, cần sắp xếp danh sách các hộp có thể mở được theo thứ tự tăng dần. Như vậy khi dùng breadth-first search sẽ đảm bảo được hộp có số nhỏ sẽ được duyệt trước nên cách mở hộp được tìm thấy đầu tiên cũng là cách có các hộp cần mở là các số nhỏ nhất.

Time complexity: O(N)

Code:

```
#include <bits/stdc++.h>

using namespace std;


// vector used dùng để đánh dấu hộp đã được duyệt

// vector previousBox dùng để lưu hộp được mở trước đó

// hộp previousBox[i] là hộp có chứa chìa khóa để mở hộp i

vector<bool> used;

vector<int> previousBox;

vector<vector<int>> adj;


void bfs() {
   queue<int> qu;
   qu.push(0);
   used[0] = true;
   previousBox[0] = -1;
   while (!qu.empty()) {
      int currentBox = qu.front();
      qu.pop();
      for (auto nextBox : adj[currentBox]) {
         if (!used[nextBox]) {
            used[nextBox] = true;
            qu.push(nextBox);
            previousBox[nextBox] = currentBox;
         }
```

```
        }
    }
}

int main() {
    int T;
    cin >> T;
    while (T--) {
        int n;
        cin >> n;
        adj = vector<vector<int>>(n);
        used = vector<bool>(n);
        previousBox = vector<int>(n);
        for (int i = 0; i < n - 1; ++i) {
            int m;
            cin >> m;
            for (int j = 0; j < m; ++j) {
                int nextBox;
                cin >> nextBox;
                --nextBox;
                adj[i].push_back(nextBox);
            }
            sort(adj[i].begin(), adj[i].end());
        }
        bfs();
        if (!used[n - 1])
            cout << -1 << '\n';
        else {
            vector<int> path;
```

```
        // duyệt lại các hộp mở được trước đó bằng vector previousBox
        for (int i = n - 1; i != -1; i = previousBox[i])
            path.push_back(i);
        reverse(path.begin(), path.end());
        cout << path.size() - 1 << '\n';
        for (int i = 0; i < path.size() - 1; ++i)
            cout << path[i] + 1 << ' ';
        cout << '\n';
      }
    }
}
```

Bài 2: Monk and the Magical Candy Bags (medium)

https://www.hackerearth.com/practice/data-structures/trees/heapspriority-queues/practice-problems/algorithm/monk-and-the-magical-candy-bags/

Monk có N cái túi đựng kẹo. Túi thứ *ith* chứa Ai viên kẹo. Sau khi nhặt một túi lên và ăn tất cả các viên kẹo trong đấy, túi sẽ tự đầy lại kẹo nhưng số lượng chỉ bằng 1 nửa so với số kẹo ban đầu. Ví dụ nhặt túi ith lên và ăn hết Ai viên kẹo trong đó, sau khi ăn hết, số kẹo trong túi ith sẽ tự hồi lại [Ai/2] viên kẹo (làm tròn xuống). Monk phải về nhà trong K phút nữa. Trong mỗi phút Monk chỉ có thể ăn hết số kẹo trong 1 túi. Xác định tối đa số kẹo Monk có thể ăn.

Input:

Dòng đầu tiên là T tương ứng là tổng số test case.

Trong mỗi test case có 2 dòng:

Dòng đầu có 2 số N và K là tổng số túi Monk có và số phút Monk phải về nhà.

Dòng tiếp theo gồm N số tương ứng là tổng số kẹo của N túi.

Output:

In ra tổng số kẹo tối đa mà Monk có thể ăn.

Điều kiện:

1 ≤ T ≤ 10

1 ≤ N ≤ 10^5

$0 \le K \le 10^5$

$0 \le Ai \le 10^{10}$

Brute-force:

Có thể thấy do mỗi phút chỉ được ăn 1 túi nên trong mỗi phút đó ta nên chọn túi có nhiều kẹo nhất và ăn hết toàn bộ số kẹo đó.

Sau khi ăn hết, update lại số kẹo trong túi do túi tự hồi lại 1 nửa của tổng số kẹo đã ăn được.

Làm tương tự trong K phút sẽ thu được kết quả cần tìm.

Time complexity: O(N*K)

Code:

```
#include <bits/stdc++.h>

using namespace std;


int main() {
   int T;
   cin >> T;
   while (T--) {
      int N, K;
      cin >> N >> K;
      vector<long long> a(N);
      for (int i = 0; i < N; ++i)
         cin >> a[i];
      long long ans = 0;
      for (int i = 0; i < K; ++i) {
         // dùng max_element để tìm phần tử lớn nhất của dãy
         vector<long long>::iterator bigCandyPosition = max_element(a.begin(), a.end());
         ans += *bigCandyPosition;
         *bigCandyPosition /= 2;
      }
```

```
        cout << ans << '\n';
    }
}
```

Improvement by using heap:

Thay vì tìm túi có số kẹo lớn nhất trong mỗi phút, dùng cấu trúc heap để lưu số kẹo của N túi.

Lấy túi có nhiều kẹo nhất chỉ tốn thời gian O(1), thêm hay loại bỏ 1 phần tử chỉ tốn O(logN).

Dùng priority_queue của C++.

Time complexity: O(K*log(N))

Code:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int T;
    cin >> T;
    while (T--) {
        int N, K;
        cin >> N >> K;
        priority_queue<long long> bags;
        for (int i = 0; i < N; ++i) {
            long long candy;
            cin >> candy;
            bags.push(candy);
        }
        long long ans = 0;
        for (int i = 0; i < K; ++i) {
            long long bigCandy = bags.top();
            bags.pop();
            ans += bigCandy;
```

```
            bigCandy /= 2;
            bags.push(bigCandy);
        }
        cout << ans << '\n';
    }
}
```

Bài 3: CLEANRBT - Cleaning Robot (medium)

https://www.spoj.com/problems/CLEANRBT/

- Một sàn nhà có kích thước WxH (W,H <=20). Trên sàn nhà có K ô gạch bị bẩn (K<=10).
- Sàn nhà được đánh dấu bằng các ký tự '.' (robot đi qua đường) và 'x' (robot không đi qua được).
- Robot có thể di chuyển nhiều lần qua cùng một ô.
- Vị trí ban đầu của robot được đánh dấu là 'o'; Vị trí viên gạch bẩn được đánh dấu '*'
- Robot chỉ có thể di chuyển theo hướng trên/dưới/trái/phải.
- Tìm số bước di chuyển nhỏ nhất của robot để làm sạch hết các ô gạch bị bẩn, hoặc in ra -1 nếu không thể làm sạch hết các ô gạch này.

1. Bruteforce
   1.1. Nhận xét
   - Lần di chuyển thứ nhất, có 4 cách di chuyển.
   - Lần di chuyển thứ 2, tương ứng với mỗi vị trí sau lần di chuyển thứ nhất, có 4 cách di chuyển => 4^2
   - Lần di chuyển tiếp theo, tương ứng với mỗi vị trí từ lần di chuyển thứ 2, có 4 cách di chuyển => 4^3
   - Ta di chuyển theo tất cả các cách cho đến khi đi qua hết cả 10 điểm bẩn hoặc đã đi hết toàn bộ sàn nhà nhưng vẫn không xóa được tất cả điểm bẩn.
   - Worst case: Mỗi lần di chuyển, robot đều đi qua tất cả các điểm nhưng không thể đi qua hết cả 10 điểm bẩn => 4^(W*H) ~ 4^400

   1.2. Cài đặt thuật toán

```
int backtrack(int r, int c, int cost, int removed){
        if(r<0||r>=H||c<0||c>=W||A[r][c]==’x’){
                return 0;
        }
        if(removed == K){
                ans=min(ans,cost);
        }
        if(cost >W*H){
                return 0;
        }
```

```
        bool removed = false;
        if(A[r][c]=='*') {
                removed = true;
                A[r][c]='.';
                removed++;
        }
        backtrack(r+1,c,cost+1,removed);
        backtrack(r-1,c,cost+1,removed);
        backtrack(r,c+1,cost+1,removed);
        backtrack(r,c-1,cost+1,removed);
        if(removed){
                A[r][c]='*';      //backtrack
        }
        return 0;
}
```

2. BFS

2.1. Nhận xét

- Dễ dàng nhận thấy thuật toán bruteforce ở trên không thể đáp ứng được yêu cầu về thời gian.
- Thuật toán bruteforce ở trên sinh ra nhiều bước đi lặp lại một cách không cần thiết (không hướng tới bất kì điểm bẩn nào).
- Ta đi tìm đường đi ngắn nhất từ điểm xuất phát của robot đi qua tất cả các điểm bẩn.
- Gọi C(i,k) là số bước robot cần để di chuyển từ vị trí điểm bẩn thứ i đến điểm bẩn thứ k; P(i) là số bước nhỏ nhất robot cần để di chuyển từ vị trí ban đầu đi qua i điểm bẩn.
- Với 10 điểm bẩn, có 10! ~ 4*1e6 cách chọn P(10). Ta cần tìm giá trị nhỏ nhất của các cách chọn này.
- Số bước để robot làm sạch i điểm bẩn = Số bước để làm sạch i – 1 điểm bẩn + số bước để đi từ điểm bẩn i-1 đến điểm bẩn i.
- min(P(i)) = min(P(i-1)+C(i,i-1)).
- Ta có thể sử dụng bfs để giải quyết bài toán này.

2.2. Cài đặt thuật toán

```
const int dr[4] = {-1, 0, 0, 1}, dc[4] = { 0,-1, 1, 0};
const int MAX = 20, STATE = 1<<10;
int dist[STATE][MAX][MAX], H, W, total;
char A[MAX][MAX+5];
int bfs(int r, int c) {
        int ans = INT_MAX;
        queue< int > q;
        dist[0][r][c] = 0;
        q.push(r), q.push(c), q.push(0);
        while(!q.empty()) {
                r = q.front(); q.pop();
                c = q.front(); q.pop();
                int mask = q.front(); q.pop();
                for(int i=0; i<4; i++) {
                        int vi = r + dr[i];
                        int vj = c + dc[i];
                        int vm = mask;
```

```
                        if(vi>=0 && vi<H && vj>=0 && vj<W && A[vi][vj]!='x') {
                                if(A[vi][vj] < total){
                                        vm |= 1<<A[vi][vj];
                                }
                                if(dist[vm][vi][vj] > dist[mask][r][c]+1) {
                                        dist[vm][vi][vj] = dist[mask][r][c] + 1;
                                        if(vm==(1<<total)-1){
                                                ans = min(ans, dist[vm][vi][vj]);
                                        }
                                        q.push(vi), q.push(vj), q.push(vm);
                                }
                        }
                }
        }
        return ((ans < INT_MAX) ? ans : -1);
}


int main() {
        int i, j, k, si, sj;
        while(scanf("%d%d", &W, &H)==2 &&(W+H)) {
                for(i=total=0; i<H; i++) {
                        scanf("%s", A[i]);
                        for(j=0; j<W; j++) {
                                if(A[i][j]=='o') si = i, sj = j;
                                else if(A[i][j]=='*') A[i][j] = total++;
                        }
                }
                for(k=0; k<(1<<total); k++)
                        for(i=0; i<H; i++)
                                for(j=0; j<W; j++)
                                        dist[k][i][j] = INT_MAX;
                printf("%d\n", bfs(si, sj));
        }
        return 0;
}
```