2.1. Avance rápido

2.1.1. Pseudocódigo y explicación

-------------------

Algoritmo Principal

-------------------

Entrada: número de casos de prueba T

Leer casosPrueba

Para cada casosPrueba:

Leer n y m

Leer la matriz de distancias[n][n]

mejorDiversidad = -1

mejorSeleccion = vector de n ceros

Para i = 0 hasta n hacer:

seleccion = voraz(n, m, distancias, i)

seleccionados = vector de n ceros

Para j = 0 hasta n

Si seleccion[j]

añadir j a seleccionar

diversidad = calcularDiversidad(seleccionados, distancias)

Si diversidad > mejorDiversidad entonces

mejorDiversidad = diversidad

mejorSeleccion = seleccion

Fin Si

Fin Para

Escribir mejorDiversidad

Escribir mejorSeleccion

Fin Para

-------------

Avance Rápido

-------------

Función voraz(n, m, distancias, inicio)

C = conjunto de candidatos

S = conjunto solución inicial

enSolucion = vector de n booleanos inicializados en falso

enSolucion[inicio] = verdadero

Mientras C no esté vacío y factible(S,m):

x = seleccionar(S, C, distancias)

Eliminar x de C

Si factible(S,m)

Añadir x a S

enSolucion[x] = verdadero

Fin Si

Fin Mientras

resultado = vector binario de tamaño n

Para i = 0 hasta n hacer:

Si enSolucion[i] entonces resultado[i] = 1

Si no, resultado[i] = 0

Fin Para

Devolver resultado

Fin Función

-----------

Seleccionar

-----------

Función seleccionar(S, C, distancias)

mejorGanancia = -1

mejorCandidato = -1

Para cada candidato en C hacer:

ganancia = 0

Para cada s en S hacer:

ganancia = ganancia + distancias[candidato][s] + distancias[s][candidato]

Fin Para

Si ganancia > mejorGanancia entonces

mejorGanancia = ganancia

mejorCandidato = candidato

Fin Si

Fin Para

Devolver mejorCandidato

Fin Función

--------

Factible

--------

Funcion factible(S,m)

Devolver (tamaño de S < m)

Este algoritmo se basa en 4 funciones:

-La función 'calcularDiversidad' suma la distancia entre cada par de elementos del subconjunto S.

Para ello se usa un doble bucle para recorrer todos los pares. En el segundo bloque se empieza

desde i para evitar contar dos veces la misma pareja de elementos, ya que sumamos ambas direcciones.

-La función 'seleccionar' selecciona el candidato en C que tenga mayor ganancia. Se usa un doble bucle

para recorrer cada candidato y, sobre cada uno, recorrer S el cual contiene las soluciones.

-La función 'factible' la cual devuelve verdadero si todavía hay elementos por seleccionar y falso si

se han alcanzado el número de elementos.

-La función 'voraz' la cual inicia la solución S con un elemento de partida (en este caso el parámetro 'inicio').

Mientras que queden candidatos en C y sea factible (todavía hay elementos por seleccionar), va seleccionando los

candidatos de mayor ganancia, los elimina de S y, si es factible, se añade en S.

Usamos un vector de booleanos, el cual contiene verdadero en i posición si este forma parte de la solución. Como

el ejercicio pide que sea un vector de 0's y 1's, lo convertimos.

'C' es el conjunto de candidatos disponibles, el cual se va actualizando eliminando el candidato elegido (así

nos aseguramos de que no se selecciona dos veces).

'S' es el conjunto solución, el cual contiene la solución actual y crece hasta alcanzar el tamaño m.

Como la solución final depende del elemento inicial escogido, recorremos cada posible inicio y seleccionamos la

solución con mayor diversidad.

2.1.2. Programación del algoritmo

#include <iostream>

#include <vector>

using namespace std;

/\*

\*Calcula la diversidad de un subconjunto

\*La función recorre todos los pares de elementos del vector 'seleccionados' y acumula la suma de sus distancias.

\*Devuelve un entero, que sería el total de las distancias.

\*/

int calcularDiversidad(const vector<int>& seleccionados, int\*\* distancias) {

int suma = 0; //entero que representa la diversidad total

for (size\_t i = 0; i < seleccionados.size(); ++i) {

//Se suman ambas direcciones de la distancia

for (size\_t j = i + 1; j < seleccionados.size(); ++j) {

suma += distancias[seleccionados[i]][seleccionados[j]] + distancias[seleccionados[j]][seleccionados[i]];

}

}

return suma;

}

/\*

\*Elige el candidato con más ganancia

\*Dado el conjunto soluciones S y el conjunto de candidatos C, la función determina que cancidato es

\*mejor sumando la ganancia de añadirlo a S

\*Devuelve el mejor candidato.

\*/

int seleccionar(const vector<int>& S, const vector<int>& C, int\*\* distancias) {

int mejorCandidato = -1; //entero que representa el mejor candidato

int mejorGanancia = -1; //entero que representa la mejor ganancia

for (int candidato : C) {

int ganancia = 0; //ganancia del candidato seleccionado

for (int s : S) {

ganancia += distancias[candidato][s] + distancias[s][candidato];

}

if (ganancia > mejorGanancia) {

mejorGanancia = ganancia;

mejorCandidato = candidato;

}

}

return mejorCandidato;

}

/\*

\*Determina si es factible seguir agregando elementos

\*La función devuelve verdadero si el numero de elementos de S es menor que m y falso si es mayor

\*/

// Función factible: true si no se ha alcanzado el tamaño m

bool factible(const vector<int>& S, int m) {

return S.size() < (size\_t)m;

}

/\*Función Avance rápido

\*A partir de un elemento de inicio, se construye la solución S agregando en cada iteración el mejor candidato

\*n es el número total de elementos y m el número de elementos a seleccionar

\*'distancias' contiene las distancias entre cada par de elementos

\*'inicio' es el elemento inicial

\*/

vector<int> voraz(int n, int m, int\*\* distancias, int inicio) {

vector<int> C; // conjunto de candidatos

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

if (i != inicio)

C.push\_back(i);

}

vector<int> S = {inicio}; // solución inicial con el elemento de inicio

vector<bool> enSolucion(n, false); //vector booleano

enSolucion[inicio] = true;

while (!C.empty() && factible(S, m)) {

int x = seleccionar(S, C, distancias); // seleccionar candidato

// Construir nuevo conjunto de candidatos sin x

vector<int> nuevoC;

for (int c : C) {

if (c != x) nuevoC.push\_back(c);

}

C = nuevoC;

// Insertar si es factible

if (factible(S, m)) {

S.push\_back(x);

enSolucion[x] = true;

}

}

// Convertir solución a vector de 0 y 1

vector<int> resultado(n);

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

resultado[i] = enSolucion[i] ? 1 : 0;

return resultado;

}

int main() {

int T;

cin >> T;

while (T--) {

int n, m;

cin >> n >> m;

// Reservar espacio para la matriz de distancias

int\*\* distancias = new int\*[n];

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

distancias[i] = new int[n];

//Lectura de matriz

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

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

cin >> distancias[i][j];

int mejorDiversidad = -1;

vector<int> mejorSeleccion;

//evaluar el algoritmo para cada posible elemento de inicio

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

vector<int> seleccion = voraz(n, m, distancias, i);

//extraer índices de elementos seleccionados

vector<int> seleccionados;

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

if (seleccion[j]) seleccionados.push\_back(j);

//calcular diversiad

int diversidad = calcularDiversidad(seleccionados, distancias);

//seleccionar mejor solución

if (diversidad > mejorDiversidad) {

mejorDiversidad = diversidad;

mejorSeleccion = seleccion;

}

}

//imprimir resultado

cout << mejorDiversidad << endl;

for (int bit : mejorSeleccion)

cout << bit << " ";

cout << endl;

//liberar memoria

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

delete[] distancias[i];

delete[] distancias;

}

return 0;

}

2.1.3. Estudio teórico

-Para la función 'calcularDiversidad' se recorren los pares distintos(i,j). Como el tamaño máximo es m,

tendría O(m²).

-Para la función 'voraz' necesitamos tener en cuenta la función 'seleccionar'. Esta recorre todos los candidatos

(en el peor de los casos hasta n-1), y para cada uno calcula su ganancia junto con los elementos acutales de la

solución (como máximo hasta m-1). Por lo tanto el orden de la función 'seleccionar' es O(n\*m).

La función 'voraz' hace eso para cada candidato. Por lo tanto, el orden de la función sería O(n\*m²) para el peor

de los casos (cuando se realizan todas las iteraciones posibles).

Para el mejor caso (número de candidatos pequeño o m pequeño), el orden sería de 0(n\*m).

2.1.4. Estudio experimental

Para realizar este estudio experimental, hemos generado diferentes matrices con distinto tamaño de n y m (hemos decidido que m = n/2).

-Primero hemos generado matrices con un n entre 5 y 15.

-Después hemos generado matrices con un n entre 15 y 30.

-Por último hemos generado matrices con un n entre 30 y 50.

Para cada una, hemos probado con diferente número de casos. En concreto, con 1000 , 5000 y 10000 casos.

Con todas estas pruebas hemos creado una gráfica la cual representa el tamaño de las entradas en el eje X y los tiempos en el eje Y.

2.1.5. Contraste estudio teórico y experimental

En la gráfica se puede ver que las entradas con distancias más grandes aumentan significativamente el tiempo de ejecución, mientras que las distancias pequeñas muestran tiempos de ejecución más bajos.

Podemos apreciar que no se aprecian discrepancias entre el estudio teórico y el experimental, por lo tanto, coinciden en ambos casos. Ambos muestran que el algoritmo es eficiente para valores pequeños de n, y que según va aumentando, el algoritmo empieza a ser más costoso.  
  
El tiempo de ejecución obtenido se debe principalmente al crecimiento cuadrático en el cálculo de la diversidad y selección de elementos. También se debe a la evaluación de cada elemento candidato con todos los elementos seleccionados actuales. Esto hace que el tiempo de ejecución del algoritmo aumente significativamente.

2.1.6. Conclusiones  
En conclusión, esta práctica nos ha ayudado sobre todo a poder comprender el tema con más facilidad ya que, el hecho de poder programar uno mismo el algoritmo nos hace tener en cuenta ciertos factores que quizá no hubiéramos tenido en cuenta solo con la teoría.   
Esto es clave para poder entender como los algoritmos se comportan en situaciones reales