# Ejercicios de Vuelta Atrás

Estructuras de Datos y Algoritmos
Universidad Complutense de Madrid (UCM)
24 pag.

# Torres de colores.

(Examen febrero 2017, sesión 2, ejercicio 3) Laura quiere construir una torre con piezas de colores. En su juego de construcciones hay piezas azules, rojas y verdes, de cada una de las cuales tiene un determinado número disponible, respectivamente $a$, $r$ y $v$. Quiere construir una torre que contenga $n >= 2$ piezas en total cada una encima de la anterior. No le gusta el color verde, asi que nunca coloca dos piezas verdes juntas, ni permite que el número de piezas verdes supere al de piezas azules en ningún momento mientras se va construyendo la torre. Además, como el color rojo es su favorito, las torres que construye siempre tienen en la parte inferior una pieza roja, y en la torre final el número de piezas rojas debe ser mayor que la suma de las piezas azules y verdes.

Implementar un algoritmo que muestre todas las formas posibles que tiene de construir una torre de la altura deseada cumpliendo con las restricciones mencionadas.

*Requisitos de implementación.*
El objetivo del problema es practicar el esquema de vuelta atrás, por lo tanto se pueden representar los colores con valores enteros. No es necesario utilizar tipos enumerados.

Para comprobar las condiciones sobre el número de piezas debes guardar en tres variables (o en un vector de 3 posiciones) el número de piezas de cada color que se han puesto en la solución que se está construyendo.

Las condiciones que se deben cumplir mientras se está construyendo la torre se deben comprobar en todas las llamdas recursivas (función `esValida`). En cambio las condiciones que debe cumplir la torre final solo se deben comprobar cuando ya se tiene una solución completa.

La salida se pide ordenada en orden lexicográfico. Para ello piensa en que orden debe el algoritmo ir probando los tres valores, de forma que las permutaciones se generen en el orden que se pide en la salida.

## Entrada

La entrada que espera el corrector automático consta de una serie de casos de prueba y acabará cuando se introduzca una línea con cuatro ceros. Cada caso de prueba se escribe en una línea y consta de 4 enteros separados por blancos. El primero es mayor que uno y representa la altura de la torre. Los tres siguientes son mayores o iguales que cero y representan los cubos de color azul, rojo y verde respectivamente.

## Salida

Para cada caso de prueba se escriben todas las posibles torres, una en cada línea ordenadas por orden lexicográfico y separando cada par de colores por un espacio. Cada caso termina con una línea en blanco. Si no se puede construir la torre con los números de bloques dados se escribirá *SIN SOLUCION*.

## Entrada de ejemplo

```
4 4 4 4
5 2 2 2
5 3 3 1
2 1 2 1
0 0 0 0
```

## Salida de ejemplo

```
rojo azul rojo rojo
rojo rojo azul rojo
rojo rojo rojo azul
rojo rojo rojo rojo

SIN SOLUCION

rojo azul azul rojo rojo
rojo azul rojo azul rojo
rojo azul rojo rojo azul
rojo azul rojo rojo verde
rojo azul rojo verde rojo
rojo azul verde rojo rojo
rojo rojo azul azul rojo
rojo rojo azul rojo azul
rojo rojo azul rojo verde
rojo rojo azul verde rojo
rojo rojo rojo azul azul
rojo rojo rojo azul verde

rojo rojo
```

```cpp
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

struct tDatos {
    int altura, numAzules, numRojos, numVerdes;
    vector<string> solucion;
    vector<int> numColores;
    bool haySolucion;
};

/* DEFINICIÓN DEL ESPACIO DE SOLUCIONES Y ÁRBOL DE EXPLORACIÓN.
* Numerando cada bloque de color de 1 a d.numColores.size(), podemos representar cada asignación de un bloque de color a una
* posición por medio de una tupla (x1, ..., xi,..., xd.altura), donde xi es un bloque de color asignado a la posición i.
* Esto corresponde a un árbol de exploración con d.altura - 1 niveles y d.numColores.size() hijos por nodo.
*/

bool esValida(const tDatos& d, int pos, int k) {
    if (pos == 2 && d.solucion[k - 1] == "verde") return false;
    else {
        if (pos == 2 && d.numColores[0] < d.numColores[2] + 1) return false;
        else return true;
    }
}

void vueltaAtras(tDatos& d, int k) {
    for (int pos = 0; pos < d.numColores.size(); pos++) {
        if (pos == 0 && d.numColores[pos] != d.numAzules || pos == 1 && d.numColores[pos] != d.numRojos || pos == 2 && d.numColores[pos] != d.numVerdes) {
            if (esValida(d, pos, k)) {
                if (pos == 0) d.solucion[k] = "azul";
                else if (pos == 1) d.solucion[k] = "rojo";
                else d.solucion[k] = "verde";
                d.numColores[pos]++;
                if (k == d.altura - 1) {
                    if (d.numColores[1] > d.numColores[0] + d.numColores[2]) {
                        d.haySolucion = true;
                        for (int pos = 0; pos < d.solucion.size(); pos++) cout << d.solucion[pos] << " "; cout << endl;
                    }
                }
                else vueltaAtras(d, k + 1);
                d.solucion[k] = " ";
                d.numColores[pos]--;
            }
        }
    }
}

void prepararVueltaAtras(tDatos& d) {
    d.solucion.resize(d.altura); d.solucion[0] = "rojo";
```

```
51      d.numColores.resize(3); d.numColores[0] = 0; d.numColores[1] = 1;
          d.numColores[2] = 0;
52      d.haySolucion = false;
53  }
54
55  bool resolverCasoDePrueba() {
56      tDatos d; cin >> d.altura >> d.numAzules >> d.numRojos >> d.numVerdes;
57      if (d.altura == 0 && d.numAzules == 0 && d.numRojos == 0 && d.numVerdes ==
          0) return false;
58
59      if (d.numRojos >= 1) {
60          prepararVueltaAtras(d);
61          vueltaAtras(d, 1);
62      }
63
64      if (!d.haySolucion || d.numRojos == 0) cout << "SIN SOLUCION" << endl;
65      cout << endl;
66
67      return true;
68  }
69
70
71  int main() {
72  #ifndef DOMJUDGE
73      std::ifstream in("AC20.in");
74      auto cinbuf = std::cin.rdbuf(in.rdbuf());
75  #endif
76
77      while (resolverCasoDePrueba());
78
79  #ifndef DOMJUDGE
80      std::cin.rdbuf(cinbuf);
81      system("PAUSE");
82  #endif
83
84      return 0;
85  }
```

**Problema número XXXXXX**

# Los funcionarios del Ministerio

**Tiempo máximo: X s Memoria máxima: X KiB**

http://www.aceptaelreto.com/problem/statement.php?id=XXXXX

El Ministro de Desinformación y Decencia se ha propuesto hacer trabajar en firme a sus funcionarios, para lo que se ha sacado de la manga una serie de trabajos (tantos como funcionarios). A pesar de su ineficacia, todos los funcionarios son capaces de hacer cualquier trabajo, aunque unos tardan más que otros. En el Ministerio todos se conocen bien, por lo que se sabe cuánto tardará cada funcionario en realizar cada uno de los trabajos. Para justificar su puesto, Su Excelencia el Sr. Ministro desea conocer la asignación óptima de trabajos a funcionarios de modo que la suma total de tiempos sea *mínima*.

## Entrada

La entrada consta de una serie de casos de prueba. Cada uno comienza con una línea con el número $N$ de funcionarios y trabajos ($1 \leq N \leq 20$). A continuación aparecerán $N$ líneas, una por funcionario, con $N$ números (entre 1 de 10.000) que indican lo que tarda ese funcionario en realizar cada uno de los $N$ trabajos.

La entrada terminará con un caso sin funcionarios, que no debe procesarse.

## Salida

Para cada caso de prueba se escribirá una línea con la suma total de tiempos que tardarán los funcionarios en realizar los trabajos asignados según la asignación óptima.

## Entrada de ejemplo

```
3
10 20 30
40 20 10
60 10 20
3
10 15 20
30 40 50
60 80 99
0
```

## Salida de ejemplo

```
30
120
```

**Autor:** Alberto Verdejo.

**Revisores:** Pedro Pablo Gómez Martín y Marco Antonio Gómez Martín.

```cpp
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

struct tDatos {
    int n;
    vector<vector<int>> matriz;
    vector<bool> marcaje;
    int tiempoOptimo;
};

/* DEFINICIÓN DEL ESPACIO DE SOLUCIONES Y ÁRBOL DE EXPLORACIÓN.
* Podemos representar cada asignación de un funcionario a un trabajo por medio de la tupla (x1, ..., xi, ..., xdatos.n),
* donde xi es un trabajo asignado al funcionario i. Esto corresponde a un árbol de exploración de datos.n niveles y donde
* cada nodo tiene datos.n hijos.
*/

void vueltaAtras(tDatos& datos, int k, int tiempoActual, const vector <int> &estimacion) {
    for (int pos = 0; pos < datos.n; pos++) {
        if (!datos.marcaje[pos]) {
            datos.marcaje[pos] = true;
            tiempoActual += datos.matriz[pos][k];
            if (k == datos.n - 1) {
                if (datos.tiempoOptimo == -1 || datos.tiempoOptimo > tiempoActual)
                    datos.tiempoOptimo = tiempoActual;
            }
            else {
                if (tiempoActual + estimacion[k + 1] < datos.tiempoOptimo || datos.tiempoOptimo == -1)
                    vueltaAtras(datos, k + 1, tiempoActual, estimacion);
            }
            datos.marcaje[pos] = false;
            tiempoActual -= datos.matriz[pos][k];
        }
    }
}

/* JUSTIFICACIÓN DE LA PODA.
* Para la optimización del problema, definiremos un vector que llevará en cada posición el mínimo de cada columna y a
* continuación, definiremos vector que llevará la suma de los mínimos desde la posición k hasta el final del vector de mínimos.
* De esta manera, tendremos una estimación menos pesimista que la de calcular el mínimo global de la matriz, y podremos
* comprobar si merece la pena seguir explorando esa rama antes de realizar la llamada al siguiente hijo.
*/

vector<int> calcularEstimacion(const tDatos& datos) {
    vector<int> estimacion(datos.n); int min;
    for (int i = 0; i < datos.n; i++) {
```

```
48          min = datos.matriz[0][i];
49          for (int j = 1; j < datos.n; j++) {
50              if (min > datos.matriz[j][i]) min = datos.matriz[j][i];
51          }
52          estimacion[i] = min;
53      }
54      for (int k = datos.n - 2; k >= 0; k--) estimacion[k] = estimacion[k + 1];
55      return estimacion;
56  }
57
58  void prepararVueltaAtras(tDatos& datos) {
59      for (int i = 0; i < datos.n; i++) {
60          vector<int> aux(datos.n);
61          for (int& j : aux) cin >> j;
62          datos.matriz.push_back(aux);
63      }
64
65      datos.marcaje.resize(datos.n);
66      for (int k = 0; k < datos.n; k++) datos.marcaje[k] = false;
67
68      datos.tiempoOptimo = -1;
69  }
70
71  bool resolverCasoDePrueba() {
72      tDatos datos; cin >> datos.n;
73      if (datos.n == 0) return false;
74
75      prepararVueltaAtras(datos);
76      vector <int> estimacion = calcularEstimacion(datos);
77      vueltaAtras(datos, 0, 0, estimacion);
78
79      cout << datos.tiempoOptimo << endl;
80
81      return true;
82  }
83
84
85  int main() {
86  #ifndef DOMJUDGE
87      std::ifstream in("AC21-22.in");
88      auto cinbuf = std::cin.rdbuf(in.rdbuf());
89  #endif
90
91      while (resolverCasoDePrueba());
92
93  #ifndef DOMJUDGE
94      std::cin.rdbuf(cinbuf);
95      system("PAUSE");
96  #endif
97
98      return 0;
99  }
```

# 25. Luces de Navidad

Mi tio se dedica a la fabricación de luces de Navidad. Cuando era pequeño me llevaba con él a la fábrica, donde se podían ver rollos y rollos de luces de colores, para adornar los árboles y las fachadas de las casas. Durante muchos años ha detectado que unas tiras se venden mas que otras. Para analizar el motivo, ha realizado un estudio de mercado con él que han detectado que a la gente no le importa realmente el color concreto de cada bombilla, sino que el aspecto total tenga bastante colorido. Por ello ha decidido hacer las tiras de luces de forma que no haya mas de dos luces seguidas del mismo color y que se cumpla que en cualquier punto de la tira la suma de las luces de un color no supere en mas de una unidad la suma de las luces de todos los demás colores. Además se debe tener en cuenta que las tiras de luces no deben consumir mas de una cierta cantidad de energía para cumplir con la legislación sobre el medio ambiente.

Ahora quiere saber cuantas posibles tiras puede hacer de una determinada longitud dado un número máximo de bombillas de cada tipo y un consuma máximo para la tira. Para poder seleccionar las tiras adecuadas cuenta con el consumo de cada tipo de bombilla.

*Requisitos de implementación.*

El problema se debe implementar empleando la técnica de vuelta atrás.

Para facilitar la implementación del programa se ha añadido al final del enunciado la salida del caso de ejemplo mostrando las posibles tiras para cada caso.

## Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de 2 líneas. En la primera se indica la longitud de la línea de luces a fabricar; el número de colores diferentes que se van a utilizar, y el consumo máximo soportado por la tira. En la siguiente se indica el consumo de cada tipo de bombilla. La entrada termina con una línea con un cero.

La longitud y el número de colores son números enteros mayores que uno. El consumo máximo es un entero mayor que cero. El consumo de cada bombilla es un valor entero mayor que cero.

## Salida

Para cada caso de prueba se escribe en una línea el número de combinaciones posibles para los valores de entrada.

## Entrada de ejemplo

```
4 2 6
2 1
4 2 5
2 1
5 3 7
2 1 3
5 3 8
2 1 3
5 3 17
5 4 3
6 4 27
6 5 4 5
0
```

## Salida de ejemplo

```
4
0
4
16
4
84
```

A continuación se muestran las combinaciones de cada caso de prueba del ejemplo para ayudar a la depuración. El primer valor es el tipo de la primera bombilla de la tira, el segundo valor es el tipo de la segunda bombilla de la tira, el tercero es el tipo de la tercera bombilla etc.

## Salida de ejemplo

```
0 1 0 1
0 1 1 0
1 0 0 1
1 0 1 0

SIN SOLUCION

0 1 0 1 1
0 1 1 0 1
1 0 0 1 1
1 0 1 0 1

0 1 0 1 0
0 1 0 1 1
0 1 1 0 0
0 1 1 0 1
0 1 1 2 1
0 1 2 1 1
1 0 0 1 0
1 0 0 1 1
1 0 1 0 0
1 0 1 0 1
1 0 1 2 1
1 0 2 1 1
1 2 0 1 1
1 2 1 0 1
2 1 0 1 1
2 1 1 0 1

1 2 1 2 2
1 2 2 1 2
2 1 1 2 2
2 1 2 1 2
```

## Salida de ejemplo

```
1 2 1 2 1 2
1 2 1 2 2 1
1 2 1 2 2 3
1 2 1 2 3 2
1 2 1 3 2 2
1 2 2 1 1 2
1 2 2 1 2 1
1 2 2 1 2 3
1 2 2 1 3 2
1 2 2 3 1 2
1 2 2 3 2 1
1 2 2 3 2 3
1 2 2 3 3 2
1 2 3 1 2 2
1 2 3 2 1 2
1 2 3 2 2 1
1 2 3 2 2 3
1 2 3 2 3 2
1 2 3 3 2 2
1 3 2 1 2 2
1 3 2 2 1 2
1 3 2 2 3 2
1 3 2 3 2 2
2 1 1 2 1 2
2 1 1 2 2 1
2 1 1 2 2 3
2 1 1 2 3 2
2 1 1 3 2 2
2 1 2 1 1 2
2 1 2 1 2 1
2 1 2 1 2 3
2 1 2 1 3 2
2 1 2 3 1 2
2 1 2 3 2 1
2 1 2 3 2 3
2 1 2 3 3 2
2 1 3 1 2 2
2 1 3 2 1 2
2 1 3 2 2 1
2 1 3 2 2 3
2 1 3 2 3 2
2 1 3 3 2 2
2 3 1 1 2 2
2 3 1 2 1 2
2 3 1 2 2 1
2 3 1 2 2 3
2 3 1 2 3 2
2 3 1 3 2 2
2 3 2 1 1 2
2 3 2 1 2 1
2 3 2 1 2 3
2 3 2 1 3 2
2 3 2 3 1 2
2 3 2 3 2 1
2 3 2 3 2 3
2 3 2 3 3 2
2 3 3 1 2 2
2 3 3 2 1 2
2 3 3 2 2 1
2 3 3 2 2 3
2 3 3 2 3 2
3 1 2 1 2 2
3 1 2 2 1 2
```





```
3 2 1 1 2 2
```

```cpp
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

struct tDatos {
    int l, numColores, consumoMaximo;
    vector<int> consumos;
    vector<int> solucion;
    vector<int> marcaje;
    int numSoluciones;
};

/* DEFINICIÓN DEL ESPACIO DE SOLUCIONES Y ÁRBOL DE EXPLORACIÓN.
* Numerando cada bombilla de 1 a datos.numColores, podemos representar cada asignación de una bombilla a una
* posición por medio de una tupla (x1, ..., xi,..., xdatos.l), donde xi es una bombilla asignada a la posición i.
* Esto corresponde a un árbol de exploración con datos.l niveles y datos.numColores hijos por nodo.
*/

bool esValida(tDatos& datos, int k, int pos, int consumoActual) {
    if (consumoActual + datos.consumos[pos] > datos.consumoMaximo) return false;
    else {
        if (k >= 2 && datos.solucion[k - 1] == datos.solucion[k - 2] && pos == datos.solucion[k - 1]) return false;
        else {
            int sumaLuces = 0;
            for (int i = 0; i < datos.numColores; i++) {
                if (i != pos) sumaLuces += datos.marcaje[i];
            }
            if (datos.marcaje[pos] > sumaLuces) return false;
            else return true;
        }
    }
}

void vueltaAtras(tDatos& datos, int k, int consumoActual) {
    for (int pos = 0; pos < datos.numColores; pos++) {
        if (esValida(datos, k, pos, consumoActual)) {
            datos.marcaje[pos]++;
            consumoActual += datos.consumos[pos];
            datos.solucion[k] = pos;
            if (k == datos.l - 1) datos.numSoluciones++;
            else vueltaAtras(datos, k + 1, consumoActual);
            datos.marcaje[pos]--;
            consumoActual -= datos.consumos[pos];
            datos.solucion[k] = -1;
        }
    }
}

void prepararVueltaAtras(tDatos& datos) {
    cin >> datos.numColores >> datos.consumoMaximo;
```

```
52
53      datos.consumos.resize(datos.numColores);
54      for (int pos = 0; pos < datos.numColores; pos++) cin >> datos.consumos
          [pos];
55
56      datos.solucion.resize(datos.l);
57      for (int pos = 0; pos < datos.l; pos++) datos.solucion[pos] = -1;
58
59      datos.marcaje.resize(datos.numColores);
60
61      datos.numSoluciones = 0;
62  }
63
64  bool resolverCasoDePrueba() {
65      tDatos datos; cin >> datos.l;
66      if (datos.l == 0) return false;
67
68      prepararVueltaAtras(datos);
69      vueltaAtras(datos, 0, 0);
70
71      cout << datos.numSoluciones << endl;
72
73      return true;
74  }
75
76
77  int main() {
78  #ifndef DOMJUDGE
79      std::ifstream in("AC23.in");
80      auto cinbuf = std::cin.rdbuf(in.rdbuf());
81  #endif
82
83      while (resolverCasoDePrueba());
84
85  #ifndef DOMJUDGE
86      std::cin.rdbuf(cinbuf);
87      system("PAUSE");
88  #endif
89
90      return 0;
91  }
```

# Compra de la semana

Alonso Rodríguez tiene que hacer la compra de la semana. Ha hecho una lista de n productos que quiere comprar. En su barrio hay m supermercados en cada uno de los cuales se dispone de todos esos productos. Pero como es un comprador compulsivo no quiere comprar más de tres productos en cada uno de los supermercados ya que así pasa más tiempo comprando (se puede suponer que $n \leq 3m$). Dispone de una lista de precios (en céntimos) de los productos en cada uno de los supermercados. Se pide diseñar un algoritmo que permita a Alonso decidir cómo hacer la compra de forma que compre todo lo que necesita y que el coste total sea mínimo.

## Entrada

La entrada comienza por una línea indicando el número de casos de prueba que deberán procesarse. Para cada caso de prueba la primera línea tiene dos números, el primero es el número de supermercados y el segundo el número de productos. Se garantiza que $0 \leq$ número de productos $\leq 3 *$ número de supermercados y que número de supermercados $\leq 20$. A continuación aparecen tantas líneas como supermercados y en cada una de las líneas el precio de todos los productos en ese supermercado. En todos los supermercados se ofrecen todos los productos.

## Salida

Por cada caso de prueba aparecerá una línea independiente con el coste de la mejor solución encontrada o bien el mensaje "Sin solucion factible" en el caso de que no haya ninguna.

## Entrada de ejemplo

```
2
6 10
1820  510  370  1000  460  324  505  640  2030  409
2000  430  450  1110  606  290  530  670  2104  501
1760  502  395  1200  550  199  525  702  1830  550
2130  640  560  1307  735  450  600  720  2150  575
1143  455  505  1140  500  400  350  550  2030  399
1200  475  403  1002  560  350  502  640  2009  460
4 1
4020
3560
5540
3540
```

## Salida de ejemplo

```
6743
3540
```

```cpp
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

struct tDatos {
    int numSupermercados, numProductos;
    vector<vector<int>> precios;
    vector<int> marcaje;
    int precioOptimo;
};

/* DEFINICIÓN DEL ESPACIO DE SOLUCIONES Y ÁRBOL DE EXPLORACIÓN.
* Podemos representar la asignación de cada supermercado a cada producto por medio de la tupla
* (x1, ..., xi, ..., xdatos.numProductos), donde xi es un supermercado asignado al producto i. Esto corresponde a un árbol de
* exploración de datos.numProductos niveles y donde cada nodo tiene datos.numSupermercados hijos.
*/

void vueltaAtras(tDatos& datos, int k, int precioActual, const vector<int> &estimacion) {
    for (int pos = 0; pos < datos.numSupermercados; pos++) {
        if (datos.marcaje[pos] <= 2) {
            datos.marcaje[pos]++;
            precioActual += datos.precios[pos][k];
            if (k == datos.numProductos - 1) {
                if (datos.precioOptimo == -1 || datos.precioOptimo > precioActual)
                    datos.precioOptimo = precioActual;
            }
            else {
                if (precioActual + estimacion[k + 1] < datos.precioOptimo || datos.precioOptimo == -1)
                    vueltaAtras(datos, k + 1, precioActual, estimacion);
            }
            datos.marcaje[pos]--;
            precioActual -= datos.precios[pos][k];
        }
    }
}

/* JUSTIFICACIÓN DE LA PODA.
* Para la optimización del problema, definiremos un vector que llevará en cada posición el mínimo de cada columna y a
* continuación, definiremos vector que llevará la suma de los mínimos desde la posición k hasta el final del vector de mínimos.
* De esta manera, tendremos una estimación menos pesimista que la de calcular el mínimo global de la matriz, y podremos
* comprobar si merece la pena seguir explorando esa rama antes de realizar la llamada al siguiente hijo.
*/

vector<int> calcularEstimacion(const tDatos& datos) {
    vector<int> estimacion(datos.numProductos); int min;
```

```
47      for (int i = 0; i < datos.numProductos; i++) {
48          min = datos.precios[0][i];
49          for (int j = 1; j < datos.numSupermercados; j++) {
50              if (min > datos.precios[j][i]) min = datos.precios[j][i];
51          }
52          estimacion[i] = min;
53      }
54      for (int k = datos.numProductos - 2; k >= 0; k--) estimacion[k] +=
          estimacion[k + 1];
55      return estimacion;
56  }
57
58  void prepararVueltaAtras(tDatos& datos) {
59      cin >> datos.numSupermercados >> datos.numProductos;
60
61      for (int i = 0; i < datos.numSupermercados; i++) {
62          vector <int> aux(datos.numProductos);
63          for (int& j : aux) cin >> j;
64          datos.precios.push_back(aux);
65      }
66
67      datos.marcaje.resize(datos.numSupermercados);
68
69      datos.precioOptimo = -1;
70  }
71
72  void resolverCasoDePrueba() {
73      tDatos datos;
74
75      prepararVueltaAtras(datos);
76      vector<int> estimacion = calcularEstimacion(datos);
77      vueltaAtras(datos, 0, 0, estimacion);
78
79      if (datos.precioOptimo != -1) cout << datos.precioOptimo << endl;
80      else cout << "Sin solucion factible" << endl;
81  }
82
83  int main() {
84  #ifndef DOMJUDGE
85      std::ifstream in("AC24-25.in");
86      auto cinbuf = std::cin.rdbuf(in.rdbuf());
87  #endif
88
89      int numCasos; cin >> numCasos;
90      for (int i = 0; i < numCasos; ++i) {
91          resolverCasoDePrueba();
92      }
93
94  #ifndef DOMJUDGE
95      std::cin.rdbuf(cinbuf);
96      system("PAUSE");
97  #endif
98
99      return 0;
100 }
```

# Blanca Navidad

Se acerca la Navidad y en los alrededores de Hill Valley ya están todas las carreteras cubiertas de nieve. El condado al que pertenece Hill Valley dispone de $m$ máquinas quitanieves para las $n(> m)$ carreteras. Pero no todas las máquinas pueden circular por todas las carreteras, pues cada máquina y cada carretera tienen una anchura y una máquina con anchura $a$ solo podrá limpiar de nieve carreteras cuya anchura sea mayor o igual que $a$. Por otra parte, cada pareja máquina-carretera tiene asociada una calidad de limpieza.

Para demostrar que eres un buen ciudadano y que te preocupa el buen uso de los impuestos que pagan tus padres, te has ofrecido a determinar, utilizando el esquema de *Vuelta atrás*, una asignación *válida* de máquinas quitanieves a carreteras que *maximice* la suma de las calidades de las carreteras que se han limpiado con las máquinas que se les ha asignado. Siempre teniendo en cuenta que cada máquina va a limpiar a lo sumo una carretera y que cada carretera se ha de limpiar a lo sumo con una máquina.

## Entrada

La entrada comienza con una línea que contiene el número de casos de prueba. La entrada de cada caso de prueba consistirá en una primera línea con los valores de $m$ y $n$, siendo $0 \leq m < n \leq 50$, una segunda línea con las anchuras de las $m$ máquinas, una tercera línea con las anchuras de las $n$ carreteras (todas las anchuras, de máquinas y carreteras, verifican $0 < a_i \leq 1000$), y finalmente $m$ líneas de $n$ números cada una que representan las calidades ($0 < c_{ij} \leq 1000$).

## Salida

Por cada caso de prueba el programa escribirá una línea con la calidad máxima obtenida.

## Entrada de ejemplo

```
3

1 2
2
1 2
7 5

2 3
2 3
4 1 5
20 1 15
15 1 5

3 6
2 3 6
2 2 5 4 8 9
10 25 6 12 20 8
5 14 10 10 13 9
16 16 17 12 11 5
```

## Salida de ejemplo

```
5
30
46
```

**Autor:** Yolanda Ortega Mallén.

```cpp
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

struct tDatos {
    int numQuitanieves, numCarreteras;
    vector<int> anchurasQuitanieves, anchurasCarreteras, marcaje;
    vector<vector<int>> calidades;
    int calidadOptima;
};

bool esValida(const tDatos& datos, int pos, int k) {
    if (datos.anchurasCarreteras[pos] >= datos.anchurasQuitanieves[k]) return true;
    else return false;
}

/* DEFINICIÓN DEL ESPACIO DE SOLUCIONES Y ÁRBOL DE EXPLORACIÓN.
* Podemos representar la asignación de cada carretera a cada quitanieves por medio de la tupla
* (x1, ..., xi, ..., xdatos.numQuitanieves), donde xi es una carretera asignada a la quitanieves i. Esto corresponde a un árbol
* de exploración de datos.numQuitanieves niveles y donde cada nodo tiene datos.numCarreteras hijos.
*/

void vueltaAtras(tDatos& datos, int k, int calidadActual, const vector <int> &estimacion) {
    if (k == datos.numQuitanieves) {
        if (datos.calidadOptima < calidadActual || datos.calidadOptima == -1)
            datos.calidadOptima = calidadActual;
    }
    else {
        for (int pos = 0; pos < datos.numCarreteras; pos++) {
            if (!datos.marcaje[pos]) {
                if (esValida(datos, pos, k)) {
                    datos.marcaje[pos] = true;
                    calidadActual += datos.calidades[k][pos];
                    if (k == datos.numQuitanieves - 1 || calidadActual + estimacion[k + 1] > datos.calidadOptima || datos.calidadOptima == -1) {
                        vueltaAtras(datos, k + 1, calidadActual, estimacion);
                    }
                    datos.marcaje[pos] = false;
                    calidadActual -= datos.calidades[k][pos];
                }
            }
        }
        if (k == datos.numQuitanieves - 1 || calidadActual + estimacion[k + 1] > datos.calidadOptima || datos.calidadOptima == -1) {
            vueltaAtras(datos, k + 1, calidadActual, estimacion);
        }
    }
}

```

```
49  /* JUSTIFICACIÓN DE LA PODA.
50  * Para la optimización del problema, definiremos un vector que llevará en cada posición el máximo de cada fila y a
51  * continuación, definiremos vector que llevará la suma de los máximos desde la posición k hasta el final del vector de máximos.
52  * De esta manera, tendremos una estimación menos pesimista que la de calcular el máximo global de la matriz, y podremos
53  * comprobar si merece la pena seguir explorando esa rama antes de realizar la llamada al siguiente hijo.
54  */
55
56  vector<int> calcularEstimacion(const tDatos& datos) {
57      vector<int> estimacion(datos.numQuitanieves); int max;
58      for (int i = 0; i < datos.numQuitanieves; i++) {
59          max = datos.calidades[i][0];
60          for (int j = 1; j < datos.numCarreteras; j++) {
61              if (max < datos.calidades[i][j]) max = datos.calidades[i][j];
62          }
63          estimacion[i] = max;
64      }
65      for (int k = datos.numQuitanieves - 2; k >= 0; k--) estimacion[k] += estimacion[k + 1];
66      return estimacion;
67  }
68
69  void prepararVueltaAtras(tDatos& datos) {
70      cin >> datos.numQuitanieves >> datos.numCarreteras;
71
72      datos.anchurasQuitanieves.resize(datos.numQuitanieves);
73      for (int pos = 0; pos < datos.numQuitanieves; pos++) cin >> datos.anchurasQuitanieves[pos];
74
75      datos.anchurasCarreteras.resize(datos.numCarreteras);
76      for (int pos = 0; pos < datos.numCarreteras; pos++) cin >> datos.anchurasCarreteras[pos];
77
78      for (int i = 0; i < datos.numQuitanieves; i++) {
79          vector<int> aux(datos.numCarreteras);
80          for (int& j : aux) cin >> j;
81          datos.calidades.push_back(aux);
82      }
83
84      datos.marcaje.resize(datos.numCarreteras);
85      for (int pos = 0; pos < datos.numCarreteras; pos++) datos.marcaje[pos] = false;
86
87      datos.calidadOptima = -1;
88  }
89
90  void resolverCasoDePrueba() {
91      tDatos datos;
92
93      prepararVueltaAtras(datos);
94      vector<int> estimacion = calcularEstimacion(datos);
95      vueltaAtras(datos, 0, 0, estimacion);
96
```

```
 97      cout << datos.calidadOptima << endl;
 98  }
 99
100  int main() {
101  #ifndef DOMJUDGE
102      std::ifstream in("AC26.in");
103      auto cinbuf = std::cin.rdbuf(in.rdbuf());
104  #endif
105
106      int numCasos; cin >> numCasos;
107      for (int i = 0; i < numCasos; ++i) {
108          resolverCasoDePrueba();
109      }
110
111  #ifndef DOMJUDGE
112      std::cin.rdbuf(cinbuf);
113      system("PAUSE");
114  #endif
115
116      return 0;
117  }
```

El 13 de Julio de 2019 se va a celebrar un concierto benéfico de rock como el que tuvo lugar hace 34 años en Wembley. Los artistas participantes ya están confirmados y solamente falta decidir el orden de actuación de los mismos. Los promotores del concierto han realizado una estimación de la cantidad de donaciones que se pueden recibir durante la actuación de cada uno de los $n$ artistas dependiendo del momento 0 a $n-1$ en el que actúan. También disponen de una tabla de "vetos" en la que cada artista ha reflejado si admite tocar o no inmediatamente después de cada uno de los demás. Por ejemplo Queene no acepta tocar después de nadie mientras que U3 acepta tocar solamente después de Chimpanzeez. Ayuda a los promotores a determinar el orden en que han de tocar los artistas para obtener la máxima donación posible según la estimación realizada.

- Implementa un algoritmo de vuelta atrás que resuelva el problema. Explica claramente los marcadores que has utilizado.
- Plantea dos posibles funciones de poda de optimalidad, razona sobre cual de ellas es mejor e impleméntala en tu algoritmo.

## Entrada

La entrada comienza con una línea que contiene el número de casos de prueba. Cada caso de prueba contendrá el valor del número de artistas $n$. A continuación figuran las estimaciones de las donaciones: una fila para cada artista. Después los vetos de los artistas: una fila para cada artista $i$ indicando si admite (1) o no (0) tocar después del artista $j$ (habrá un 0 en la posición $i$).

## Salida

Por cada caso de prueba el programa escribirá una línea con la donación máxima estimada (suma de las donaciones obtenidas por cada artista en el momento que le corresponde tocar). En caso de que no sea posible satisfacer los vetos se escribirá `NEGOCIA CON LOS ARTISTAS`.

## Entrada de ejemplo

```
2
3
10 20 30
140 20 10
160 10 20
0 1 1
0 0 1
0 0 0
3
10 20 30
140 20 10
160 10 20
0 0 1
0 0 1
0 0 0
```

## Salida de ejemplo

```
210
NEGOCIA CON LOS ARTISTAS
```

```cpp
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

struct tDatos {
    int n;
    vector<vector<int>> donaciones, vetos;
    vector<bool> marcaje;
    int donacionOptima;
};

bool esValida(const tDatos& datos, int pos, int k, int artistaAnterior) {
    if (k == 0 || k != 0 && datos.vetos[pos][artistaAnterior] == 1) return
      true;
    else return false;
}

/* DEFINICIÓN DEL ESPACIO DE SOLUCIONES Y ÁRBOL DE EXPLORACIÓN.
* Podemos representar la asignación de cada artista a cada posición por medio
  de la tupla (x1, ..., xi, ..., xdatos.n),
* donde xi es un artista asignado a la posición i. Esto corresponde a un árbol
   de exploración de datos.n niveles
* y donde cada nodo va decreciendo en datos.n - 1 hijos por nivel.
*/

void vueltaAtras(tDatos& datos, int k, int artistaAnterior, int
  donacionActual, const vector<int> &estimacion) {
    for (int pos = 0; pos < datos.n; pos++) {
        if (!datos.marcaje[pos]) {
            if (esValida(datos, pos, k, artistaAnterior)) {
                datos.marcaje[pos] = true;
                donacionActual += datos.donaciones[pos][k];
                if (k == datos.n - 1) {
                    if (datos.donacionOptima < donacionActual)
                        datos.donacionOptima = donacionActual;
                }
                else {
                    if (donacionActual + estimacion[k + 1] >
                    datos.donacionOptima)
                        vueltaAtras(datos, k + 1, pos, donacionActual,
                    estimacion);
                }
                datos.marcaje[pos] = false;
                donacionActual -= datos.donaciones[pos][k];
            }
        }
    }
}

/* JUSTIFICACIÓN DE LA PODA.
* Para la optimización del problema, definiremos un vector que llevará en cada
   posición el máximo de cada columna y a
* continuación, definiremos vector que llevará la suma de los máximos desde la
   posición k hasta el final del vector de máximos.
* De esta manera, tendremos una estimación menos pesimista que la de calcular
```

```
     el máximo global de la matriz, y podremos
 49  * comprobar si merece la pena seguir explorando esa rama antes de realizar la
     llamada al siguiente hijo.
 50  */
 51
 52  vector<int> calcularEstimacion(const tDatos& datos) {
 53      vector<int> estimacion(datos.n); int max;
 54      for (int i = 0; i < datos.n; i++) {
 55          max = datos.donaciones[0][i];
 56          for (int j = 0; j < datos.n; j++) {
 57              if (max < datos.donaciones[j][i]) max = datos.donaciones[j][i];
 58          }
 59          estimacion[i] = max;
 60      }
 61      for (int k = datos.n - 2; k >= 0; k--) estimacion[k] += estimacion[k + 1];
 62      return estimacion;
 63  }
 64
 65  void prepararVueltaAtras(tDatos& datos) {
 66      cin >> datos.n;
 67
 68      for (int i = 0; i < datos.n; i++) {
 69          vector<int> aux(datos.n);
 70          for (int& j: aux) cin >> j;
 71          datos.donaciones.push_back(aux);
 72      }
 73
 74      for (int i = 0; i < datos.n; i++) {
 75          vector<int> aux(datos.n);
 76          for (int& j : aux) cin >> j;
 77          datos.vetos.push_back(aux);
 78      }
 79
 80      datos.marcaje.resize(datos.n);
 81      for (int pos = 0; pos < datos.n; pos++) datos.marcaje[pos] = false;
 82
 83      datos.donacionOptima = -1;
 84  }
 85
 86  void resolverCasoDePrueba() {
 87      tDatos datos;
 88
 89      prepararVueltaAtras(datos);
 90      vector<int> estimacion = calcularEstimacion(datos);
 91      vueltaAtras(datos, 0, 0, 0, estimacion);
 92
 93      if (datos.donacionOptima != -1) cout << datos.donacionOptima << endl;
 94      else cout << "NEGOCIA CON LOS ARTISTAS" << endl;
 95  }
 96
 97  int main() {
 98  #ifndef DOMJUDGE
 99      std::ifstream in("ECONT-AC06.in");
100      auto cinbuf = std::cin.rdbuf(in.rdbuf());
101  #endif
102
```

```
103     int numCasos; cin >> numCasos;
104     for (int i = 0; i < numCasos; ++i) {
105         resolverCasoDePrueba();
106     }
107 
108 #ifndef DOMJUDGE
109     std::cin.rdbuf(cinbuf);
110     system("PAUSE");
111 #endif
112 
113     return 0;
114 }
```