Fundamentos de Algoritmia

Examen de enero Curso 2022/2023

NOMBRE: .....

Normas de realización del examen

1. Debes programar soluciones para cada uno de los tres ejercicios, probarlas y entregarlas en el juez automático accesible en la dirección `http://exacrc`. Para la entrega el juez sólo tiene los datos de prueba del enunciado del problema.

2. Escribe comentarios que expliquen tu solución, justifiquen por qué se ha hecho así y ayuden a entenderla. Calcula la complejidad de todas las funciones que implementes.

3. En el juez te identificarás con el nombre de usuario y contraseña que has recibido al comienzo del examen. El nombre de usuario y contraseña que has estado utilizando durante la evaluación continua **no** son válidos.

4. Escribe tu **nombre y apellidos** en un comentario en la primera línea de cada fichero que subas al juez.

5. Tus soluciones serán evaluadas por el profesor independientemente del veredicto del juez automático. Para ello, el profesor tendrá en cuenta **exclusivamente** el último envío que hayas realizado de cada ejercicio.

Ejercicio 1

El dueño de la frutería de mi barrio está muy preocupado porque el margen de beneficio del negocio es cada vez más pequeño. Lleva meses apuntando cada día lo que gasta en Mercamadrid para adquirir la fruta por la mañana y el importe de lo que vende a lo largo del día. Sabiendo el beneficio que tiene cada día (diferencia entre lo que gana y lo que gasta), ¿podrías calcular la longitud de la secuencia más larga de días consecutivos en los que ha estado perdiendo más de una cierta cantidad? También queremos conocer cuándo comenzó esa *mala racha* y cuál es la máxima pérdida producida durante esos días.

Se pide:

1. Especifica una función que reciba un vector con el beneficio de cada día y calcule la longitud de la secuencia más larga de días consecutivos en los que ha estado perdiendo más de una cierta cantidad.
2. Implementa una función que resuelva el problema pedido, devolviendo los tres datos que se indican en el enunciado. El coste de la implementación debe ser lineal en el número de días.
3. Indica un invariante que permita probar la corrección del algoritmo implementado y justifica que el coste del algoritmo es lineal en el número de días.

Entrada

La entrada consiste en una serie de casos de prueba. Cada caso comienza con una línea en que se indica el número de días de los que se tiene apuntado el beneficio y el límite de la pérdida que se quiere permitir. En la línea siguiente aparece el beneficio de cada día. La entrada termina con un valor 0 que no debe procesarse.

El número de días es mayor que 0 y menor que 100 000. El límite de la pérdida que se quiere permitir es un número negativo en el intervalo [-1000,-1]. El beneficio es un número entero en el intervalo $[-1000, 1000]$.

Salida

Para cada caso de entrada se escribirá el máximo número de días consecutivos en los que se ha perdido más de la cantidad dada, seguido del día en que se comenzó a perder más de esa cantidad y del máximo que se ha perdido durante esos días. Si existen varias rachas con el número máximo de días se elegirá aquella que tenga una pérdida mayor y si existen varias con el número máximo de días y la misma pérdida máxima se elegirá aquella que ocurra en último lugar. Los días comienzan a numerarse en el cero.

Si no existe ningún día en que se hayan producido pérdidas mayores al valor indicado se escribirá `SIN PERDIDAS`.

Entrada de ejemplo

```
9 -5
-7 -8 3 5 -2 -10 -7 -6 -5
4 -4
-4 -4 -4 -4
3 -2
3 -1 3
8 -1
2 -3 -5 -2 -1 -4 -10 -2
8 -1
2 -3 -5 -2 -1 -5 -2 -2
0
```

Salida de ejemplo

```
3 5 -10
SIN PERDIDAS
SIN PERDIDAS
3 5 -10
3 5 -5
```

Esquema de solución

Se trata de un problema del tipo "segmento de longitud máxima". La especificación de la función que devuelve la longitud del segmento máximo es:

P:$\{v.size \geq 0\}$
`f(vector<int> v, int k) dev int lmax`
Q:$\{lmax = \max p,q : (0 \leq p \leq q \leq v.size \wedge \forall i : p \leq i < q : v[i] < k) : q - p\}$

La especificación del comienzo de la racha y el mínimo valor en ella se podría escribir de manera análoga, pero formalizar todos los detalles en este caso resulta farragoso.

La idea es recorrer el vector de pérdidas de izquierda a derecha, manteniendo en el invariante la información correspondiente a la peor racha encontrada hasta el momento (el predicado anterior reemplazando `v.size` por i, siendo i la posición que se está explorando). Además, para poder actualizar correctamente esta información también es necesario que en el invariante también esté disponible la información correspondiente a la última racha, la que termina en la posición i.

```
int longMax = 0, longAct = 0; // longitud del segmento máximo encontrado hasta i
int longAct = 0; //  longitud del segmento que acaba en i
int maxi = v[0]; // perdida maxima en el segmento maximo encontrado hasta i
int maxiAct = v[0]; // perdida maxima en el segmento que acaba en i
int iniMax = 0; // inicio del segmento maximo encontrado hasta i
int iniAct = 0; // inicio del segmento que acaba en i
for (int i = 0; i < v.size(); ++i) {
    if (v[i] <= k) { // sigue el segmento
        ++longAct;
        if (v[i] < maxiAct) maxiAct = v[i];
        if (longMax < longAct) { // Comprueba si se ha encontrado un segmento mayor
            longMax = longAct;
            iniMax = iniAct;
            maxi = maxiAct;
        }
        else if (longMax == longAct && maxi < maxiAct) { // igual longitud se
comparan las perdidas maximas
            maxi = maxiAct; iniMax = iniAct;

        }
    }
    else { // se rompe el segmento
        longAct = 0;
        iniAct = i+1;
        maxiAct = v[i];
    }
}
```

Ejercicio 2

Los polinomios de Hermite son un ejemplo de polinomios ortogonales que encuentran su principal ámbito de aplicaciones en la mecánica cuántica. Se definen mediante la ecuación de recurrencia:

$$H_n(y) = \begin{cases} 1 & n = 0 \\ 2y & n = 1 \\ 2yH_{n-1}(y) - 2(n-1)H_{n-2}(y) & n > 1 \end{cases}$$

Se pide implementar una función recursiva eficiente que calcule el valor del n-ésimo polinomio de Hermite para un cierto valor de y.

- Explica el algoritmo empleado e impleméntalo.
- Plantea la recurrencia de la solución implementada e indica su coste.

Entrada

La entrada consta de una serie de casos. Cada caso se escribe en una línea y consiste en un número n $(0 \leq n \leq 10)$ seguido del valor de y en el que se quiere $(1 \leq y \leq 20)$. La entrada termina con el valor -1.

Salida

Para cada valor de entrada mostrar en una línea el valor del polinomio pedido.

Entrada de ejemplo

```
0 3
1 6
2 4
3 2
10 20
-1
```

Salida de ejemplo

```
1
12
62
40
9906193528929760
```

Esquema de solución

Se puede realizar una solución final o no final del algoritmo.

La idea es semejante al algoritmo de obtener en n-esimo valor de la sucesión de Fibonacci.

Solución no final, se recibe el valor de n y él de y y se devuelven dos valores, correspondientes al valor de los polinómios H_{n-1} y H_{n-2}. Con estos dos valores se calcula el valor del polinomio H_n en el punto y utilizando la recurrencia indicada en el enunciado.

```
using lli = long long int;
struct tSol {
    lli H1,H2;
};

tSol resolver(int n, int x) {
    if (n == 0) return {1,0};
    else if (n == 1) return {2*x, 1};
```

```
    else {
        tSol s = resolver(n-1,x);
        return {2*x*s.H1 - 2*(n-1)* s.H2, s.H1};
    }
}
```

Solución final. El valor pedido se va calculando en los parámetros. `h1` lleva el valor del polinomio H_n y `h2` lleva el valor del polinomio H_{n-1}.

```
using lli = long long int;

lli resolver(int k,int n, int x, lli h1, lli h2) {
    if (k == n) return h1;
    else {
        return resolver(k+1,n,x,2*x*h1 - 2*k* h2,h1);
    }
}

bool resuelveCaso() {
    int num, x;
    std::cin >> num;
    if (num == -1) return false;
    std::cin >> x;
    if (num == 0) std::cout << "1\n";
    else if (num == 1) std::cout << 2*x << '\n';
    else {
        lli s = resolver(1,num,x,2*x,1);
        std::cout << s << "\n";
    }
    return true;
}
```

Ejercicio 3

Dado un grafo valorado no dirigido $G = (V, A)$ y un entero r, escribir un algoritmo de vuelta atrás que encuentre, si existe, un clique de tamaño r y de *mínimo* coste en G. Un clique es un subgrafo completo, es decir, un subconjunto $V' \subseteq V$ tal que existe una arista entre cada par de vértices en V'. El tamaño de un clique es su número de vértices y su coste es la suma de los costes de sus aristas.

El algoritmo debe seleccionar entre los vértices V del grafo los r que van a pertenecer al clique. Un vértice puede pertenecer al clique si está conectado con todos los otros vértices del clique. Ten en cuenta que, durante la exploración, aquellas ramas en las que el número de vértices sin explorar no sea suficiente para completar el clique deberán ser podadas.

Se pide:

1. Describir el árbol de exploración y la forma del vector solución que se emplean en la solución del problema.
2. Implementar el algoritmo de vuelta atrás que resuelve el problema.
3. Piensa también en una estimación del coste que se pueda utilizar: aunque no la implementes, explícala.

Entrada

La entrada comienza con una línea que contiene el número de casos de prueba. Cada caso de prueba contendrá inicialmente el número de vértices n del grafo ($0 < n < 10$) y el tamaño r del clique ($0 < r \leq n$). A continuación, en una matriz simétrica de dimensión n por n se indica el coste (estrictamente positivo) de la arista entre los vértices correspondientes, o bien un 0 si no son adyacentes.

Salida

Para cada caso de prueba, si existe un clique del tamaño pedido el programa escribirá el coste mínimo y a continuación, separados por blancos, los vértices que lo componen. En caso contrario se escribirá `NO EXISTE`. Estudia el orden en que se deben explorar las ramas del árbol de exploración para que si dos cliques tienen el mismo coste el programa muestre el que tenga el vértice con menor número en primer lugar; si es el mismo, se quedará con el de menor valor en la segunda posición, etc.

Entrada de ejemplo

```
3
3 2
0 1 2
1 0 3
2 3 0
4 4
0 1 2 3
1 0 4 5
2 4 0 6
3 5 6 0
4 3
0 1 0 0
1 0 0 0
0 0 0 2
0 0 2 0
```

Salida de ejemplo

```
1 0 1
21 0 1 2 3
NO EXISTE
```

Esquema de solución

Se trata de un problema de optimización.

Este problema se resuelve utilizando para representar la solución un vector $sol[0..n-1]$ cuyo tamaño es el número n de vértices del grafo y de modo que $sol[i] \in \{true, false\}$ indica si el vértice i forma parte del clique. El árbol de exploración es un árbol binario, donde la rama izquierda significa que el vértice de ese nivel se selecciona y la rama derecha corresponde a no seleccionar el vértice. Es importante darse cuenta que, una vez se seleccione el vértice r-ésimo del clique, no hay que seguir explorando en el árbol por debajo sino que para todos los vértices restantes habrá que completar el vector solución con $sol[i] = 0$.

Marcaje. Se utilizará una variable entera *numVertices* para llevar la cuenta del número de vértices que ya se han seleccionado; cuando *numVertices* sea r se tendrá el clique y habrá que parar la búsqueda en profundidad por ese vértice. También se utilizará otra variable *costeAct* con el coste de las aristas correspondientes a los vértices seleccionados hasta el momento.

```
using matriz = std::vector<std::vector<int>>;

bool esValida (int V, int r, int numVertices,matriz const& costes, int k, std::vector<
    bool> & sol) {
    // No le quedan suficientes vértices para completar
    if (V - k < r - numVertices) return false;
    // Esta conectado con todos los vértices añadidos hasta este momento
    int i = 0;
    while (i < k && (!sol[i] || costes[k][i] != 0)) ++i;
    return i == k;
}

// Sumar el coste de todas las aristas que conectan con los vertices ya puestos
int calcularCoste(matriz const& costes, int k, std::vector<bool> & sol){
    int i = 0; int suma = 0;
    while (i < k) {
        if (sol[i]) suma += costes[k][i];
        ++i;
    }
    return suma;
}

void VA (int V, int r, matriz const& costes, int k, int& costeAct, int numVertices, std::
    vector<bool> & sol, int& costeMejor, std::vector<bool>& solMejor, bool & existe){

    // Selecciono el vértice k
    sol[k] = true;
    ++numVertices;
    if (esValida(V,r,numVertices,costes,k,sol)){ // Está conectado con todos los otros
    vértices del clique
        int costeAristas = calcularCoste(costes,k,sol);
        costeAct += costeAristas;
        if (numVertices == r) { // es solucion
            if (costeAct < costeMejor) {
                costeMejor = costeAct;
                solMejor = sol;
                // Completa la solución con falsos
                for (int i = k+1; i < V; ++i) solMejor[i] = false;
                existe = true;
            }
        }
```

```
        else if (k < V-1) // Todavia quedan vertices por explorar
            VA(V, r, costes, k+1, costeAct, numVertices, sol, costeMejor,  solMejor,
    existe);
        costeAct -= costeAristas;
    }
    --numVertices;

    // No cojo el vertice
    sol[k] = false;
    // siempre es valido
    // Nunca es solucion valida porque no tiene exactamente r vertices
    if (k < V-1)  // Todavia quedan vertices por explorar
        VA(V, r, costes, k+1, costeAct, numVertices, sol, costeMejor,  solMejor, existe);
}


void resuelveCaso() {
// lectura de los datos
    int V,r;
    std::cin >> V >> r;
    matriz costes(V,std::vector<int>(V));
    int suma = 1; // Para inicializar el coste mejor
    for (int i = 0; i < V; ++i)
        for (int j = 0; j < V; ++j) {
            std::cin >> costes[i][j];
            suma += costes[i][j];
        }
    // Inicializar parámetros
    int costeAct = 0;
    int numVertices = 0;
    std::vector<bool> sol(V, false);
    int costeMejor = suma;
    std::vector<bool> solMejor(V);
    bool existe = false;
    VA (V, r, costes, 0, costeAct, numVertices, sol, costeMejor,  solMejor, existe);
    if (existe) {
        std::cout << costeMejor ;
        for (int i = 0; i < V; ++i)
            if (solMejor[i]) std::cout << '␣' << i;
        std::cout << '\n';
    }
    else std::cout << "NO␣EXISTE\n";
}
```

Se podría obtener una estimación que nos ayude a poder ramas del árbol a partir del valor mínimo de todas las aristas del grafo. Así, si se han escogido ya vértices (es decir, el valor de usados es), lo mejor que podría ocurrir es que el coste de cada una de las aristas que faltan por añadir sea . Este número se obtiene restando a todas las aristas que existen entre r vértices las que existen entre i, con lo que

$$estimacion = coste - actual + (r * (r+1)/2 - i * (i+1)/2) * m$$