UNIVERSIDAD NACIONAL DE SAN ANTONIO ABAD DEL CUSCO

FACULTAD DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA, INFORMÁTICA Y MECÁNICA

ESCUELA PROFESIONAL DE INGENIERÍA INFORMÁTICA Y DE SISTEMAS

ACM CHAPTER CUSCO

CONCURSO DE PROGRAMACIÓN

CUSCONTEST XIX

PROBLEMSET CON SOLUCIONES

Cusco, 4 de Agosto de 2023

Este problemset contiene 14 problemas etiquetados de la ‘A’ a la ‘N’.

Información General

A menos que se indique lo contrario, las siguientes condiciones son válidas para todos los problemas.

Nombre del programa

1. La solución debe ser enviada en formatos del lenguaje seleccionado. Ejm: codigo.c, codigo.cpp, codigo.java, codigo.py, codigo.cs.

Entrada

1. La entrada debe ser leída desde la entrada estándar (consola).
2. La entrada consiste en un único caso de prueba, que es descrito en el formato de cada problema. No existen datos extras en la entrada.
3. Cuando una línea de datos contiene muchos valores, estos son separados por exactamente un espacio entre ellos. No existen otros espacios en las entradas.
4. Se utiliza el alfabeto Inglés. No hay letras con tildes, diéresis, eñes, u otros símbolos.

Salida

1. La salida debe ser escrita como salida estándar (consola).
2. El resultado debe ser escrito en la cantidad de líneas especificada para cada problema. No debe imprimirse otros datos. Ejm: no incluir: "ingrese el número".
3. Cuando una línea de datos de salida contiene muchos valores, estos deben ser separados por exactamente un espacio entre ellos. No deben imprimirse otros espacios en las salidas.
4. Debe ser utilizado el alfabeto Inglés. No letras con tildes, diéresis, eñes, u otros símbolos.

Límite de Tiempo

1. El límite de tiempo informado para cada problema corresponde con el tiempo total permitido para la ejecución completa de los casos de prueba.

Consejos

1. Para leer múltiples números en una línea en Python usa: A = [int(x) for x in input().split(' ')]
2. Para soluciones en java, enviar el archivo .java sin el "package name".

Problemas coordinados por Jared León, y planteados por:

Autor	Afiliación
Dennis Huilca	Senior Software Engineer, InnovarITP, PE
Edú Sanchez	Senior Software Engineer, Google, BR
Grover Castro	PhD in Computer Sc., Universität Leipzig, DE
Isaac Campos	Tema lead, Finbe USA, USA
Jared León	PhD Stud. in Maths, University of Warwick, UK
John Vargas	Senior Data Scientist, Topaz, USA
Josué Nina	Data integration and ETL developer, Provista, USA
Rodolfo Quispe	Applied Data Scientist, Microsoft, USA

Problema A. El profundo impacto de Emma

Archivo de entrada:	`Entrada estándar`
Archivo de salida:	`Salida estándar`
Límite de tiempo:	1 segundo
Límite de memoria:	64 megabytes
Autor:	`John Vargas`

La profesora Emma llevó a sus n alumnos de jardín a visitar el hoyo natural más profundo de la tierra. La profesora hablaba vigorosamente por el celular mientras los niños observaban el pozo muy de cerca hasta que... Oh no! ahora solo hay $n-1$ niños, y no hay rastro del alguno más! La profesora entra en pánico, pero rápidamente se tranquiliza al pensar que seguramente el niño fue corriendo de vuelta a la ciudad por haber olvidado su billetera o algo parecido (eso explicaría que los otros niños se encuentren en shock).

La profesora inicialmente tenía a todos los alumnos numerados del 1 al n, y ahora decide averiguar cuál es el niño que le falta. Ayúdala con esta tarea.

Entrada

La primera línea contiene un número entero t ($1 \leq t \leq 100$), indicando el número de casos de prueba.

Cada caso de prueba comienza con un entero n ($2 \leq n \leq 10^4$). La siguiente línea contiene $n-1$ enteros distintos a_i ($1 \leq a_i \leq n$) separados por un espacio y en ningún orden en particular, indicando la lista de niños que todavía están presentes.

Salida

Para cada caso de prueba, imprime el número del niño desaparecido.

Ejemplo

Entrada estándar	Salida estándar
2 8 1 2 3 5 6 7 8 2 1	4 2

Solución

Conocimientos requeridos: Matemática básica.

Dados $a_1, \ldots, a_{n-1}$, si el número del niño que desaparece es x, se sabe que $a_1 + \cdots + a_{n-1} + x = 1 + 2 + \cdots + n = n(n+1)/2$, por lo que $x = n(n+1)/2 - (a_1 + \cdots + a_{n-1})$.

Esta expresión puede ser calculada en $O(n)$, teniendo la solución una complejidad final de $O(tn)$.

Implementación en C++:

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

int main(){
  int te;
  cin >> te;
  while (te--) {
    int n;
    cin >> n;
    int sum = n * (n + 1) / 2;
    for (int i = 0; i < n - 1; i++) {
      int x; cin >> x;
      sum -= x;
    }
    cout << sum << "\n";
  }
  return 0;
}
```

Problema B. El juego de la vida de Conway

Archivo de entrada:	`Entrada estándar`
Archivo de salida:	`Salida estándar`
Límite de tiempo:	1 segundo
Límite de memoria:	64 megabytes
Autor:	`Dennis Huilca`

En esta ocasión deberás escribir un simulador del Juego de la Vida de Conway en un toro. Es decir, un tablero de $n \times n$ cuya parte superior está conectada a su parte inferior y cuyas laterales también están conectadas entre sí. Una posicion del tablero tiene exactamente 4 vecinos: los que comparten un lado en común.

Se da una regla de actualización para celdas muertas y otra para celdas vivas. Estas reglas son dadas en forma de una cadena con 5 caracteres que contienen los símbolos "*" y ".". Para una celda muerta, si el i-ésimo caracter de su regla correspondiente está vivo y exactamente $i - 1$ de sus vecinos están vivos, entonces esta celda comienza a vivir en la siguiente generación. Para una celda viva, si el i-ésimo caracter de su regla correspondiente está vivo y exactamente $i - 1$ de sus vecinos están vivos, entonces esta celda continúa viva en la siguiente generación.

Por ejemplo, si la regla de actualización para celdas muertas es "..*.." y para celdas vivas es "**...", entonces una celda muerta comienza a vivir si y solo si tiene dos vecinos vivos, y una celda viva se mantiene viva si y solo si tiene cero o un vecino vivo.

Calcula la evolución del juego por m generaciones.

Entrada

La primera línea contiene un número entero t ($1 \leq t \leq 10$), indicando el número de casos de prueba.

Cada caso de prueba comienza con dos líneas: la regla de actualización para celdas muertas y la regla de actualización para celdas vivas respectivamente. Ambas líneas contienen 5 símbolos que son "*" o ".". La tercera línea contiene dos enteros n ($2 \leq n \leq 25$) y m ($1 \leq m \leq 10^3$). Cada una de las siguientes n líneas contiene n caracteres indicando el estado inicial del tablero. Una celda viva es denotada por "*" y una muerta por ".".

Salida

Para cada caso de prueba, imprime n líneas indicando el estado final del tablero, cada una conteniendo los símbolos "*" o ".".

Ejemplo

Entrada estándar	Salida estándar
2 ..*.. **... 3 100 ... .*. ... .**.. .**.. 4 1 .... *... *...	... .*. ... *... **.* **.* *...

Solución

Conocimientos requeridos: Arreglos multi-dimensionales (matrices).

El problema requiere una simulación de las reglas m veces. La implementación es sencilla pero relativamente larga.

Complejidad $O(tmn^2)$.

Implementación en Python:

```
import copy
def actualizar(tabla):
    for i in range(N):
        for j in range(N):
            if l[i][j] == 0:
                if (l[i-1][j] + l[i][j-1] + l[(i+1)%N][j] + l[i][(j+1)%N
]) in vivoMuert:
                    tabla[i][j] = 1
                else:
                    tabla[i][j] = 0
            else:
                if (l[i-1][j] + l[i][j-1] + l[(i+1)%N][j] + l[i][(j+1)%N
]) in vivoReg:
                    tabla[i][j] = 1
                else:
                    tabla[i][j] = 0
    return tabla

for _ in range(int(input())):
    regla_muerto = input()
    regla_vivo = input()
    vivoReg = [-1]*5
    vivoMuert = [-1]*5
    for i in range(5):
        vivoReg[i] = i if regla_vivo[i] == '*' else -1
        vivoMuert[i] = i if regla_muerto[i] == '*' else -1
    N,M = list(map(int,input().split()))
    l = [[0] for i in range(N)]
    for i in range(N):
        l[i] = list(map(int,list([('1' if x == '*' else '0') for x in
input()])))
    for _ in range(M):
        l = actualizar(copy.deepcopy(l))
    for i in range(N):
        print (''.join(map(str, [('*' if x == 1 else '.') for x in l[i
]])))
```

Problema C. Los recortes de Mordor

Archivo de entrada:	`Entrada estándar`
Archivo de salida:	`Salida estándar`
Límite de tiempo:	1 segundo
Límite de memoria:	64 megabytes
Autor:	`Jared León`

El mayor enemigo de Sauron, Gándalf se encuentra alterando las tierras de Mordor para tener mayor ventaja estratégica en la batalla de Morannon. Mordor puede ser visto como un rectángulo de lados x e y con el mismo número de dígitos. Gándalf decidió que la mayor ventaja real la tendrá cuando el área de Mordor sea la mínima posible. Usando la ayuda de los Ents, Gándalf puede modificar x e y de la siguiente manera peculiar. Él puede seleccionar la misma posición en la escritura de ambos dígitos e intercambiarlos. Por ejemplo, si $x = 3251$ e $y = 8373$, Gándalf puede seleccionar la tercera posición, modificando los números a 3271 y 8353 respectivamente. Él puede realizar este cambio las veces que sean convenientes (incluso 0 veces). Ayúdalo a encontrar el área mínima de Mordor que puede alcanzarse realizando estos cambios.

Entrada

La primera línea contiene un número entero t ($1 \leq t \leq 100$), indicando el número de casos de prueba.

Cada caso de prueba contiene dos enteros x ($1 \leq x \leq 10^{10^4}$) e y ($1 \leq y \leq 10^{10^4}$), ambos con el mismo número de dígitos.

Salida

Para cada caso de prueba, calcula el área mínima requerida. Como este valor puede ser muy grande, imprímelo módulo $10^9 + 7$.

Ejemplo

Entrada estándar	Salida estándar
1 25 41	945

Solución

Conocimientos requeridos: Álgebra elemental y descomposición polinómica de un número.

Sea $n = 1 + \lfloor \log_{10} x \rfloor$ el número de dígitos de x e y. Sean también x_i e y_i los i-ésimos dígitos más bajos de x e y respectivamente, con indexación 0. Luego

$$\begin{aligned} xy &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} x_i y_j 10^{i+j} \\ &= \sum_{i=0}^{n-1} x_i y_i 10^{2i} + \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} (x_i y_j + x_j y_i)\, 10^{i+j}. \end{aligned}$$

Es fácil ver que $\sum_{i=0}^{n-1} x_i y_i 10^{2i}$ no cambia con las operaciones de Gándalf. Por lo tanto, el objetivo es minimizar $\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} (x_i y_j + x_j y_i)\, 10^{i+j}$. Para analizar si un intercambio de dígitos vale la pena, se puede observar que

$$(x_i y_j + x_j y_i) - (x_i y_i + x_j y_j) = x_i\,(y_j - y_i) + x_j\,(y_i - y_j) = (x_j - x_i)\,(y_i - y_j)\,.$$

Por lo tanto, si $(x_j - x_i)$ y $(y_i - y_j)$ tienen el mismo signo (ambos positivos o ambos negativos), entonces $x_i y_j + x_j y_i < x_i y_i + x_j y_j$. Si tienen signos opuestos (uno positivo y el otro negativo), entonces $x_i y_i + x_j y_j < x_i y_j + x_j y_i$.

Entonces, para minimizar xy es conveniente siempre dejar para toda posición i, que x_i sea el menor de x_i e y_i; y que y_i sea el mayor.

Luego de esto, la respuesta puede ser computada calculando $\sum_{i=0}^{n-1} x_i 10^i \mod 10^9 + 7$ y $\sum_{i=0}^{n-1} y_i 10^i \mod 10^9 + 7$ separadamente y luego calculando el producto de ambos módulo $10^9 + 7$.

La complejidad por caso de prueba es $O(n)$, dejando una complejidad total de $O(tn)$.

Implementación en C++:

```cpp
#include <bits/stdc++.h>
using namespace std;
const int mod = (int)1e9 + 7;
typedef long long int ll;

int main(){
  int te; cin >> te;
  while (te--) {
    string x, y; cin >> x >> y;
    int n = x.size();
    for (int i = 0; i < n; ++i)
      if(x[i] < y[i]) swap(x[i], y[i]);
    ll xx = 0, yy = 0;
    for (int i = 0; i < n; ++i) {
      xx = (xx * 10 + x[i] - '0') % mod;
      yy = (yy * 10 + y[i] - '0') % mod;
    }
    cout << (xx * yy) % mod << "\n";
  }
  return 0;
}
```

Problema D. El camarada Bourdukovsky

Archivo de entrada:	`Entrada estándar`
Archivo de salida:	`Salida estándar`
Límite de tiempo:	1 segundo
Límite de memoria:	64 megabytes
Autor:	`Edu Sánchez`

Estalló la guerra! El camarada Z. Bourdukovsky está a la búsqueda de soldados jóvenes pero no hay suficientes reclutas voluntarios. Por este motivo, el camarada recurre al Gulag más poblado de Kasrilevka para reclutar a los reclusos más jóvenes.

Él tiene una lista de los nombres de los n prisioneros $s_1, s_2, \ldots, s_n$ junto con el año de su nacimiento $a_1, a_2, \ldots, a_n$. Lo que decide hacer primero es ordenar su lista del más viejo al más joven, y en caso de colisión de años de nacimiento, alfabéticamente de menor a mayor. Desafortunadamente, la única computadora funcional del Gulag no tiene Microsoft Excel instalado, por lo que el camarada pide tu ayuda para ordenar su lista según sus requerimientos. Escribe un programa para él.

Entrada

La primera línea contiene un número entero t ($1 \le t \le 100$), indicando el número de casos de prueba.

Cada caso de prueba comienza con un entero n ($1 \le n \le 5 \times 10^3$). Luego, cada una de las siguientes n líneas contiene un número a_i ($10^3 \le a_i \le 10^5$) y una cadena s_i ($1 \le |s_i| \le 50$) sin espacios ni símbolos especiales y cuya primera letra es mayúscula, siendo todas las demás minúsculas. Ambos están separados por un espacio simple.

Salida

Para cada caso de prueba, imprime los nombres dados en el orden requerido.

Ejemplo

Entrada estándar	Salida estándar
2 6 1985 Karataev 1965 Butkovsky 1977 Govorov 2005 Ruth 2005 Glina 2005 Yanet 2 1929 Krashnoshtanov 1944 Pecinovsky	Butkovsky Govorov Karataev Glina Ruth Yanet Krashnoshtanov Pecinovsky

Solución

Conocimientos requeridos: Ordenamiento.

Se puede almacenar a cada persona en un arreglo o vector, guardando la información como un par número-cadena (como en un pair<int, string> en C++ o una túpla o lista en Python). Al ordenar tal arreglo, la mayoría de los lenguajes ordena por el primer elemento y luego por el segundo. Por lo tanto, es suficiente ordenar tal arreglo y mostrar los nombres en tal orden.

La complejidad por caso de prueba es $O(n \log n)$, dejando una complejidad total de $O(tn \log n)$.

Implementación en C++:

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

int main(){
  int te;
  cin >> te;
  while (te--) {
    int n;
    cin >> n;
    vector<pair<int, string>> a(n);
    for(auto &[x, y]: a) cin >> x >> y;
    sort(a.begin(), a.end());
    for(auto [x, gg]: a)
      cout << gg << "\n";
  }
  return 0;
}
```

Problema E. El popular Lucho

Archivo de entrada:	`Entrada estándar`
Archivo de salida:	`Salida estándar`
Límite de tiempo:	0.5 segundos
Límite de memoria:	64 megabytes
Autor:	`Grover Castro`

Lucho es el estudiante más popular del Cusco, es por eso que Movistar lo tiene muy vigilado. Movistar mantiene una lista de los amigos de Lucho y las antenas telefónicas que están muy cerca de sus casas (cada amigo está asociado a exactamente una antena). Algunas de estas antenas están conectadas alámbricamente por m cables. Para que Lucho se comunique con uno de sus amigos, debe existir una secuencia de antenas conectadas por cables que comience con la antena de Lucho y termine con la antena del amigo en cuestión. Dada la configuración de las instalaciones, Lucho puede comunicarse con todos sus $n-1$ amigos. Sin embargo, ya que Lucho también reúne a las personas, la oficina de Movistar dedicada a Lucho está interesada en saber cuántos pares de amigos de Lucho pueden comunicarse entre sí. Acabas de entrar a hacer tus prácticas en Movistar, así que estás encargado de hacer un programa que calcule esta información.

Entrada

La primera línea contiene un número entero t ($1 \leq t \leq 100$), indicando el número de casos de prueba.

Cada caso de prueba comienza con dos enteros n ($2 \leq n \leq 10^4$) y m ($1 \leq m \leq 4 \times 10^6$). Cada una de las siguientes líneas contiene dos enteros a_i y b_i indicando que hay una conexión bidireccional entre las antenas a_i y b_i. Lucho siempre tiene la antena 1, y siempre hay una secuencia de antenas conectadas desde la antena 1 a cualquier otra antena.

Salida

Para cada caso de prueba, imprime el número de pares de amigos de Lucho que pueden comunicarse mutuamente.

Ejemplo

Entrada estándar	Salida estándar
2 5 6 2 1 1 3 1 4 2 3 3 4 5 4 2 1 1 2	6 0

Ejemplo 1: Estos son todos los pares de amigos (antenas) de Lucho que pueden comunicarse entre sí:

2 - 4: A través de la secuencia de antenas $(2, 1, 4)$.

2 - 5: A través de la secuencia de antenas $(2, 3, 4, 5)$.

3 - 4: A través de la secuencia de antenas $(3, 4)$.

3 - 5: A través de la secuencia de antenas $(3, 2, 1, 4, 5)$.

2 - 3: A través de la secuencia de antenas $(2, 3)$.

4 - 5: A través de la secuencia de antenas $(4, 5)$.

Solución

Conocimientos requeridos: Teoría de grafos básica y combinatoria.

Dado que Lucho siempre puede comunicarse con todas las otras personas, es fácil ver que se tiene un grafo conexo. Por lo que cada par de vértices puede conectarse entre sí. Existen $\binom{n}{2} = n(n-1)/2$ pares de vértices, siendo ésta la respuesta siempre.

Implementación en C++:

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

int main(){
  int te;
  cin >> te;
  while (te--) {
    int n, m;
    cin >> n >> m;
    while (m--) {
      int a, b;
      cin >> a >> b;
    }
    cout << (n - 1) * (n - 2) / 2 << "\n";
  }
  return 0;
}
```

Problema F. Cifrado I

Archivo de entrada:	`Entrada estándar`
Archivo de salida:	`Salida estándar`
Límite de tiempo:	1 segundo
Límite de memoria:	64 megabytes
Autor:	`Isaac Campos`

Olvidaste la contraseña de tu cuenta bancaria, pero recuerdas que cada vez que cambias de contraseña, guardas la nueva clave cifrada en un papel dentro de un cajón.

La forma de cifrado que usas es bastante particular. Una contraseña cifrada consta de tres pilas de números. Para reconstruirla tienes que extraer un número del tope de alguna de las tres pilas y añadirlo al final de la contraseña que estás reconstruyendo. Luego solo tienes que repetir este proceso hasta que no te queden números en las pilas.

Siendo un gran amante del orden y la lexicografía, sabes que todas las contraseñas que creaste formaran el número más pequeño posible después de descifrarlo.

Para poder acceder a tu cuenta, tendrás que descifrar todas tus contraseñas pasadas, por lo que prefieres crear un programa que las descifre por ti.

Entrada

La primera línea contiene un número entero t ($1 \leq t \leq 100$), indicando el número de casos de prueba.

Para cada una de las tres pilas, se da un número p_i ($1 \leq p_i \leq 10$) en una línea, indicando el número de caracteres en la línea siguiente, los cuales son dados separados por un espacio simple, estando el último número en el tope de la pila. Se garantiza que cada contraseña contiene dígitos distintos entre sí.

Salida

Para cada caso de prueba, imprime la contraseña descifrada.

Ejemplo

Entrada estándar	Salida estándar
2 7 3 4 5 6 7 8 9 1 1 1 2 3 3 9 2 3 4 1 7 3 6 5 8	129876543 271485693

Ejemplo 1: Seleccionas el tope de la segunda pila “1” y lo concatenas al final de la contraseña (que por el momento está vacía). Luego seleccionas el tope de la tercera pila, “2”, y lo concatenas al final de la contraseña. Ahora tu contraseña es “12”, ahora que la segunda y tercera pila están vacías no te queda más que extraer uno a uno los elementos de la primera pila para terminar de formar la contraseña “129876543”.

Solución

Conocimientos requeridos: Arreglos.

Ya que los caracteres serán siempre dígitos, es óptimo siempre el menor de todos los disponibles hasta agotar las tres pilas.

Complejidad final: $O(tp_i)$.

Implementación en C++:

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

int main(){
  int te; cin >> te;
  while (te--) {
    int na; cin >> na;
    vector<int> a(na);
    for(auto &e: a) cin >> e;
    int nb; cin >> nb;
    vector<int> b(nb);
    for(auto &e: b) cin >> e;
    int nc; cin >> nc;
    vector<int> c(nc);
    for(auto &e: c) cin >> e;
    na--; nb--; nc--;
    while(na + nb + nc > -3){
      if(na >= 0 && (nb < 0 || (nb >= 0 && a[na] < b[nb])) && (nc < 0 || (nc >= 0 && a[na] < c[nc]))){
        cout << a[na--];
      } else if (nb >= 0 && (na < 0 || (na >= 0 && b[nb] < a[na])) && (nc < 0 || (nc >= 0 && b[nb] < c[nc]))) {
        cout << b[nb--];
      } else{
        cout << c[nc--];
      }
    }
    cout << "\n";
  }
  return 0;
}
```

Problema G. Cifrado II

Archivo de entrada:	`Entrada estándar`
Archivo de salida:	`Salida estándar`
Límite de tiempo:	1 segundo
Límite de memoria:	64 megabytes
Autor:	`Isaac Campos`

Desde la última vez que perdiste tu contraseña aprendiste un par de cosas sobre seguridad y resulta que cifrar tus contraseñas en 3 pilas no es la forma más segura del mundo. Por este motivo, ahora decidiste cifrar tu contraseña en n pilas utilizando un proceso parecido. Esta vez únicamente tomas el número menor de entre todas las pilas.

Una vez más perdiste la contraseña de tu cuenta para el CusContest, *-no estamos seguros de cómo estás participando-*, pero igualmente guardaste todas las contraseñas que tuviste cifradas. Dada la lista de contraseñas cifradas, descífralas para poder entrar al concurso.

Entrada

La primera línea contiene un número entero t ($1 \leq t \leq 100$), indicando el número de casos de prueba.

Cada caso de prueba comienza con un entero n ($1 \leq n \leq 10^3$), indicando el número de pilas. Para cada una de las pilas, el número de elementos de tal pila p_i se da en una línea, y en la siguiente se dan los elementos de la pila separados por un espacio. Se garantiza que el total de los números de las n pilas son distintos entre sí y que $\sum_{i=1}^{n} p_i \leq 10^3$.

Salida

Para cada caso de prueba, imprime la contraseña descifrada.

Ejemplo

Entrada estándar	Salida estándar
2 2 3 1 3 5 2 2 4 4 3 3 6 2 3 10 8 7 1 4 5 1 5 9 11 12	42531 246378101211951

Solución

Conocimientos requeridos: Arreglos.

Los límites de este problema impiden verificar todas las pilas cada vez, por lo que otra estrategia es necesaria. Es posible mantener una cola de prioridad (montículo) que siempre indique la pila con el menor elemento en el tope. Luego, cada vez que un elemento es extraído, se puede actualizar el montículo con el tope de la última pila usada.

Complejidad final: $O(t \sum_{i=1}^{n} p_i \log \sum_{i=1}^{n} p_i)$.

Implementación en C++:

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

int main() {
  int t;
  cin >> t;
  while (t--) {
    int n; cin >> n;
    stack<int> S[n];
    for (int i = 0; i < n; i++){
      int ss; cin >> ss;
      for (int j = 0; i < ss; j++){
        int a; cin >> a;
        S[i].push(a);
      }
    }
    priority_queue<pair<int,int>> Q;
    for (int i = 0; i < n; i++)
      Q.push({-S[i].top(), i});
    while (!Q.empty()) {
      pair<int, int> curr = Q.top();
      Q.pop();
      cout << -curr.first;
      S[curr.second].pop();
      if(!S[curr.second].empty())
        Q.push({-S[curr.second].top(), curr.second});
    }
    cout << "\n";
  }
  return 0;
}
```

Problema H. Ahora soy formal!

Archivo de entrada:	`Entrada estándar`
Archivo de salida:	`Salida estándar`
Límite de tiempo:	1.5 segundos
Límite de memoria:	64 megabytes
Autor:	`Jared León`

Ahora trabajas para SUNAT! Llegó la hora de alegrar algunos negocios pequeños. Te encuentras en una calle con n tiendas cuyo capital es de $t_1, t_2, \ldots, t_n$. Lo que harás será seleccionar un rango contíguo de tiendas (sabes que las tiendas más pequeñas suelen estar agrupadas) y pedirles a todas las tiendas en el rango una "contribución" de p soles para no inspeccionarlas más detalladamente. Debes seleccionar p de tal forma que cada tienda tenga la capacidad de pagar dicha cantidad. Y obviamente, quieres maximizar la cantidad recolectada. Por fortuna también eres un programador y puedes automatizar esta tarea.

Entrada

La primera línea contiene un número entero t ($1 \leq t \leq 10$), indicando el número de casos de prueba.

La primera línea de cada caso de prueba contiene un entero n ($1 \leq n \leq 3 \times 10^3$). La siguiente línea contiene n enteros t_i ($1 \leq p_i \leq 10^5$) separados por un espacio.

Salida

Para cada caso de prueba, imprime la máxima cantidad que puedes recolectar de las tiendas.

Ejemplo

Entrada estándar	Salida estándar
2 6 2 4 3 9 4 9 6 154 4 4 9 4 9	15 154

Solución

Conocimientos requeridos: Arreglos y prefijos.

Es fácil ver que si el rango a tomar es $[i, j]$, entonces siempre es óptimo escoger $p = \text{mín}\{t_i, t_{i+1}, \ldots, t_j\}$, y la solución será $p(j - i + 1)$. Si para cada posible rango, se calcula el mínimo en tal rango, entonces es fácil escoger aquel que maximice esta expresión.

El problema ahora se reduce a calcular el mínimo en un rango, para todos los $O(n^2)$ rangos. Usar una estructura de datos como un Segment Tree (árbol de segmentos) o BIT (árbol binario indexado) resultará en un Tiempo Límite Excedido. Ya que se quiere calcular este mínimo para todos los rangos, es posible reutilizar información. Si se tiene el mínimo para el rango $[i, j - 1]$, entonces calcular el mínimo para el rango $[i, j]$ en tiempo constante es trivial.

La complejidad por caso de prueba es $O(n^2)$, dejando una complejidad total de $O(tn^2)$.

Implementación en C++:

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

int main(){
  int te; cin >> te;
  while (te--) {
    cin >> n;
    vector<int> a(n);
    for(auto &e: a) cin >> e;
    int gg = 0;
    for (int i = 0; i < n; ++i)
      int mini = a[i];
      for (int j = i; j < n; j++) {
        mini = min(mini, a[j]);
        gg = max(gg, mini * (j - i + 1));
      }
    cout << gg << "\n";
  }
  return 0;
}
```

Problema I. Pillao Matao?

Archivo de entrada:	`Entrada estándar`
Archivo de salida:	`Salida estándar`
Límite de tiempo:	1 segundo
Límite de memoria:	64 megabytes
Autor:	`Josué Nina`

En el pueblo de Pillao Matao, el alcalde decidió regalar cuchillos conmemorativos a los visitantes en puestos instalados a lo largo de la carretera principal. Estos puestos, se colocaron en los kilómetros que corresponden a múltiplos de x y también en kilómetros correspondientes a múltiplos de y. Los números x e y son primos. En cada puesto se regala exactamente un cuchillo por persona. Tú acabas de llegar al pueblo y tu objetivo es coleccionar la máxima cantidad de cuchillos antes de que llegues al kilómetro n, donde tendrás que ponerlos a prueba...

¿Cuál es la máxima cantidad de cuchillos que puedes coleccionar?

Entrada

La primera línea contiene un número entero t ($1 \leq t \leq 10^5$), indicando el número de casos de prueba.

Cada caso de prueba contiene los enteros n ($1 \leq n \leq 10^7$), x ($1 \leq x \leq 10^2$), e y ($1 \leq y \leq 10^2$) separados por un espacio.

Salida

Para cada caso de prueba, imprime el número de cuchillos que podrás coleccionar.

Ejemplo

Entrada estándar	Salida estándar
3 10 3 5 10 2 7 5 7 11	4 5 0

Solución

Conocimientos requeridos: Principio de inclusión-exclusión.

Existen $\lfloor (n-1)/x \rfloor$ múltiplos de x y $\lfloor (n-1)/y \rfloor$ múltiplos de y menores a n. Si se suman estas cantidades se cuentan los múltiplos de x e y dos veces, por lo que es necesario restar esta cantidad. Ya que x e y son números primos, el mínimo común múltiplo de estos es xy, por lo que existen $\lfloor (n-1)/(xy) \rfloor$ múltiplos de xy menores que n.

La respuesta es $\lfloor (n-1)/x \rfloor + \lfloor (n-1)/y \rfloor - \lfloor (n-1)/(xy) \rfloor$.

La complejidad por caso de prueba es $O(1)$, dejando una complejidad total de $O(t)$.

Implementación en C++:

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long int ll;

int main(){
  int te;
  cin >> te;
  while (te--) {
    ll n, x, y;
    cin >> n >> x >> y;
    n--;
    ll gg = n / x + n / y - n / (x * y);
    cout << gg << "\n";
  }
  return 0;
}
```

Problema J. The Beatles

Archivo de entrada:	`Entrada estándar`
Archivo de salida:	`Salida estándar`
Límite de tiempo:	1 segundo
Límite de memoria:	64 megabytes
Autor:	`Jared León`

Tres escarabajos se encuentran en las coordenadas $(0, 0)$, $(0, 1)$, y $(1, 0)$ del plano $\mathbb{Z}^2$. Los escarabajos consideran una configuración estable si cada uno de ellos está a distancia exactamente 1 de otro escarabajo y sus tres posiciones no son colineales (en particular, su posición inicial es estable).

Solamente un escarabajo puede moverse al mismo tiempo y al final de su movimiento, la configuración de estos debe ser estable. ¿Cuál es el mínimo número de movimientos que necesitan para llegar a la configuración de coordenadas (x_1, y_1), (x_2, y_2), y (x_3, y_3)?

Entrada

La primera línea contiene un número entero t ($1 \leq t \leq 10$), indicando el número de casos de prueba.

Cada caso de prueba contiene seis enteros x_1, y_1, x_2, y_2, x_3, y_3 ($|x_i|, |y_i| \leq 25$). Se garantiza que la configuración final es estable. Es fácil probar que siempre es posible que los escarabajos lleguen a esta posición.

Salida

Para cada caso de prueba, calcula el mínimo número de movimientos que necesitan para llegar a su posición final.

Ejemplo

Entrada estándar	Salida estándar
1 2 2 3 2 2 1	4

Solución

Conocimientos requeridos: Caminos mínimos en grafos sin peso (BFS) y álgebra lineal básica.

Por conveniencia, llamaremos a las configuraciones estables simplemente configuraciones. La siguiente observación es de utilidad para resolver este problema.

Proposición. *Dos configuraciones son diferentes si y solo si las suma de los radiovectores de los escarabajos en ambas configuraciones son diferentes.*

Demostración. Para la condición necesaria, probaremos que una configuración determina únicamente la suma de los radiovectores de las posiciones. Sea (x, y) la posición del escarabajo central en una configuración y las posiciones de los otros escarabajos $(x + \Delta x, y)$ y $(x, y + \Delta y)$. La suma de los radiovectores de estas posiciones es $(3x + \Delta x, 3y + \Delta y)$.

Para la condición suficiente, sea v la suma de los radiovectores de alguna configuración. No es difícil ver que la posición del escarabajo central es siempre round$(v/3)$ (donde el redondeo se aplica independientemente a ambas coordenadas del vector). Teniendo la posición de este escarabajo, es fácil calcular las posiciones de los demás escarabajos (por ejemplo, si esta posición es (x, y) y si la coordenada en x de v es $3x - 1$ entonces hay un escarabajo a la izquierda, si es $3x + 1$ hay un escarabajo a la derecha). Por lo tanto, la configuración se determina únicamente. □

Entonces, el problema se puede replantear como: pasar de la posición $(1, 1)$ a la posición $(x_1 + x_2 + x_3, y_1 + y_2 + y_3)$ saltando de posición en posición de acuerdo a las reglas de movimiento en el mínimo número de movimientos.

Es posible tratar las posiciones del nuevo problema como vértices de un grafo, donde el número de vecinos de un vértice es una constante (¿cuál?). El número de vértices y aristas de este grafo es $O(\max x_i \max y_i)$ Los límites del problema hacen posible usar el algoritmo BFS (búsqueda en anchura) en este grafo para encontrar la longitud de un camino más corto de la posición inicial a la posición final.

La complejidad por caso de prueba es $O(\max x_i \max y_i)$, dejando una complejidad total de $O(t \max x_i \max y_i)$.

Es posible realizar una solución $O(1)$ por caso de prueba observando un patrón para posiciones pequeñas y probando que tal patrón se mantiene por inducción.

Implementación en C++:

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

vector<pair<int, int>> adyacentes(int x, int y){
  vector<pair<int, int>> gg;
  int cx = round(x / 3.0), cy = round(y / 3.0);
  int ax = cx + 1, ay = cy;
  if(x == 3 * cx - 1) ax = cx - 1;
  int bx = cx, by = cy + 1;
  if(y == 3 * cy - 1) by = cy - 1;
  // Mover el centro hacia la otra ubic.
  int mx = ax != cx ? ax:bx, my = ay != cy ? ay:by;
  gg.push_back({mx + ax + bx, my + ay + by});
  // Mover a hacia la ubic. 1/3
  mx = ax == cx - 1 ? (cx + 1):(cx - 1);
  my = ay;
  gg.push_back({mx + cx + bx, my + cy + by});
```

```cpp
  // Mover a hacia la ubic. 2/3
  my = by;
  gg.push_back({mx + cx + bx, my + cy + by});
  // Mover a hacia la ubic. 3/3
  mx = ax;
  gg.push_back({mx + cx + bx, my + cy + by});
  // Mover b hacia la ubic. 1/3
  mx = bx;
  my = by == cy - 1 ? (cy + 1):(cy - 1);
  gg.push_back({mx + ax + cx, my + ay + cy});
  // Mover b hacia la ubic. 2/3
  mx = ax;
  gg.push_back({mx + ax + cx, my + ay + cy});
  // Mover b hacia la ubic. 3/3
  my = by;
  gg.push_back({mx + ax + cx, my + ay + cy});
  return gg;
}

int main(){
  int te; cin >> te;
  while (te--) {
    int x1, y1, x2, y2, x3, y3;
    cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;
    int x = x1 + x2 + x3, y = y = y1 + y2 + y3;

    // BFS
    queue<pair<int, int>> q;
    q.push({1, 1});
    map<pair<int, int>, int> d;
    d[{1, 1}] = 0;
    while (q.size() > 0) {
      pair<int, int> p = q.front(); q.pop();
      int vx = p.first, vy = p.second;
      int dist = d[p];
      for (auto [ux, uy]: adyacentes(vx, vy)) {
        if(d.find({ux, uy}) == d.end() || dist + 1 < d[{ux, uy}]){
          d[{ux, uy}] = dist + 1;
          q.push({ux, uy});
        }
        // Objetivo localizado
        if(ux == x && uy == y){
          queue<pair<int, int>> empty;
          swap(q, empty);
          break;
        }
      }
    }
    cout << d[{x, y}] << "\n";
  }
  return 0;
}
```

Problema K. Chusky el viajante I

Archivo de entrada:	`Entrada estándar`
Archivo de salida:	`Salida estándar`
Límite de tiempo:	1 segundo
Límite de memoria:	64 megabytes
Autor:	`Rodolfo Quispe`

Chusky quiere viajar por todos los países de Sudamérica. Él no sabe mucho sobre esta parte del mundo y quiere que lo ayudes a planear su viaje. La primera parte es determinar, cuánto tiempo necesitará para visitar todos los paises de este continente. Él planea estar 2 meses en cada país. Imprime el número de meses que necesita para viajar por toda Sudamérica.

Entrada

Este problema no tiene entrada.

Salida

Imprime la respuesta pedida.

Ejemplo

Entrada estándar	Salida estándar
	24

Solución

Conocimientos requeridos: Imprimir en consola.

Este problema no necesita explicación.

Implementación en Python:

```
print(24)
```

Problema L. Chusky el viajante II

Archivo de entrada:	`Entrada estándar`
Archivo de salida:	`Salida estándar`
Límite de tiempo:	1 segundo
Límite de memoria:	64 megabytes
Autor:	`Rodolfo Quispe`

Ahora que Chusky sabe cuántos meses estará en Sudamérica, él necesita calcular cuánto dinero necesitará para su viaje. Él pasará por n países, en los cuales el cambio de moneda del i-ésimo país es c_i por 1 dogcoin (la moneda del país de Chusky), y la cantidad que necesitará en tal país es d_i en moneda local. Ayuda a chusky a determinar cuánto dinero necesita en dogcoins.

Entrada

La primera línea contiene un número entero t ($1 \leq t \leq 100$), indicando el número de casos de prueba.

Cada caso de prueba comienza con un entero n ($1 \leq n \leq 10^4$). Luego, cada una de las siguientes n líneas contiene los números c_i ($1 \leq c_i \leq 10^5$) y d_1 ($0 \leq d_i \leq 10^5$) separados por un espacio. Se garantiza que d_i es un múltiplo de c_i.

Salida

Para cada caso de prueba, imprime la cantidad de dinero requerida.

Ejemplo

Entrada estándar	Salida estándar
2 5 5 20 4 20 10 0 1 15 3 12 2 100 100 1 100	28 101

Solución

Conocimientos requeridos: Imprimir en consola.

Obviamente la cantidad de dogcoins que necesita Chusky para el país i es d_i/c_i, lo cual está garantizado de ser entero. La cantidad total necesaria es la suma de esta expresión para todo i.

La complejidad por caso de prueba es $O(n)$, dejando una complejidad total de $O(tn)$.

Implementación en C++:

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

int main(){
  int te;
  cin >> te;
  while (te--) {
    int n;
    cin >> n;
    int gg = 0;
    while (n--) {
      int ci, di;
      cin >> ci >> di;
      gg += di / ci;
    }
    cout << gg << "\n";
  }
  return 0;
}
```

Problema M. Chusky el viajante III

Archivo de entrada:	`Entrada estándar`
Archivo de salida:	`Salida estándar`
Límite de tiempo:	1 segundo
Límite de memoria:	64 megabytes
Autor:	`Rodolfo Quispe`

Chusky comenzó su viaje y está encantado con Sudamérica. Ahora está en Cusco y mientras caminaba por la Plaza de Armas conoció a Jarencio. Intercambiaron números de teléfono y ahora están conversando por WhatsApp. Chusky percibe algo peculiar en Jarencio: cuando él le escribe algún mensaje que contiene alguna vocal, Jarencio no responde e ignora su mensaje. Jarencio es bastante excéntrico y piensa que las vocales no son caracteres cool. Chusky quiere preguntar algo a Jarencio urgentemente pero tiene problemas escribiendo sin vocales. Dado el mensaje original de Chusky, transfórmalo para un mensaje que sea cool para Jarencio.

Entrada

La primera línea contiene un número entero t ($1 \leq t \leq 100$), indicando el número de casos de prueba.

Cada caso de prueba contiene una cadena de texto s ($1 \leq |s| \leq 10^4$) en una única línea. La cadena s puede únicamente contiene letras minúsculas y espacios.

Salida

Para cada caso de prueba, imprime el texto que Jarencio aprobaría, con las palabras separadas *por un único espacio*.

Ejemplo

Entrada estándar	Salida estándar
2 quiero que me recomiendes un lugar para ir a comer	qr q m rcmnds n lgr pr r cmr

Solución

Conocimientos requeridos: Manipulación de cadenas de texto.

Cada palabra debe ser tratada independientemente debido a los espacios. Luego es suficiente quitar las vocales e imprimir la lista de palabras que no sean vacías (palabras como "a") se convierten en una cadena vacía.

La complejidad por caso de prueba es $O(|s|)$, dejando una complejidad total de $O(t|s|)$.

Implementación en Python:

```python
def quitar_vocales(s):
    return ''.join([x for x in s if x not in "aeiou"])

for _ in range(int(input())):
    a = [quitar_vocales(x) for x in input().split(' ')]
    a = [i for i in a if i != ""]
    print(' '.join(a))
```

Problema N. Palíndromo

Archivo de entrada:	`Entrada estándar`
Archivo de salida:	`Salida estándar`
Límite de tiempo:	1 segundo
Límite de memoria:	64 megabytes
Autor:	`Jared León`

Dadas n cadenas de texto $s_1, \ldots, s_n$, se quiere formar un palíndromo (una cadena que se escribe de igual de derecha a izquierda como de izquierda a derecha) concatenando dichas cadenas las veces que sea necesario. La única condición es que usar la cadena s_i tiene un costo de c_i por uso. Es decir, que el costo de usar la cadena s_i cinco veces, es $5c_i$. Determina el costo mínimo de formar un palíndromo o reporta que es imposible hacerlo.

Entrada

La primera línea contiene un número entero t ($1 \leq t \leq 10$), indicando el número de casos de prueba.

La primera línea de cada caso de prueba contiene un entero n ($1 \leq n \leq 40$). La siguiente línea contiene n enteros c_i ($1 \leq c_i \leq 10^9$) separados por un espacio. Cada una de las siguientes n líneas contiene una cadena de texto s_i ($1 \leq |s_i| \leq 20$).

Salida

Para cada caso de prueba, en caso de ser posible construir un palíndromo con las cadenas disponibles, imprime el costo mínimo de hacerlo. De otra forma, imprime el texto "Imposible".

Ejemplo

Entrada estándar	Salida estándar
2 3 3 4 5 ba abc cbaa 2 1 2 abcd abc	7 Imposible

Solución

Conocimientos requeridos: Caminos mínimos en grafos con peso (Dijkstra), prefijos y sufijos.

Aunque aparentemente la cantidad de cadenas que es necesario verificar es infinita, a veces es útil pensar en las propiedades matemáticas de algoritmos que nunca terminan.

Supongamos un algoritmo hipotético que intenta construir la cadena final s concatenando las cadenas dadas desde ambos extremos hacia el centro, probando potencialmente infinitas configuraciones. Durante su ejecución, supongamos que tal algoritmo encontró el prefijo $p = s_{p_1} s_{p_2} \dots s_{p_x}$ y el sufijo $q = s_{q_y} s_{q_{y-1}} \dots s_{q_1}$. Si el algoritmo colocó cada cadena sin permitir que existan colisiones y suponiendo sin pérdida de generalidad que $|p| \leq |q|$, entonces los últimos $|p|$ caracteres de q forman p (es decir, p es un sufijo de q). Por lo tanto, $q = r \cdot p$. Más aun, si suponemos que el algoritmo sólo concatena una nueva cadena con el de menor longitud entre p y q, entonces r es el prefijo de alguna cadena dada. Este estado del algoritmo puede ser representado por el par ordenado $(\emptyset, r)$ (donde usamos el símbolo $\emptyset$ para representar la cadena vacía). Luego, si es posible, el algoritmo usará una cadena s_i cuyo prefijo es r (para no generar una colisión) y pagará el costo c_i. En el caso en que $|r| \leq |s_i|$, es decir que $s_i = r \cdot t$, entonces el nuevo estado del algoritmo puede ser representado por $(t, \emptyset)$. De otra forma, sabemos que $r = u \cdot s_i$, y el nuevo estado del algoritmo puede ser representado por $(\emptyset, u)$.

Con tal representación de los estados de este algoritmo hipotético, es posible observar que siendo $s = \max s_i$, solamente existen a lo más $2ns$ posibles estados a los que el algoritmo puede llegar (pero infinitos prefijos y sufijos posibles). Por este motivo, es posible tratar el conjunto

$$\{(\emptyset, x) : x \text{ es prefijo de algún } s_i\} \cup \{(x, \emptyset) : x \text{ es sufijo no vacío de algún } s_i\}$$

como el conjunto de vértices de un grafo G cuyas aristas están dadas por la posibilidad del algoritmo hipotético de moverse de la representación de un estado al otro y cuyo peso es el costo de usar la cadena que hace posible dicho movimiento. El número de vértices de G es $O(ns)$ y cada vértice tiene grado $O(n)$, por lo que el número de aristas es $O(n^2 s)$. Inicialmente, el algoritmo se encuentra en el vértice $(\emptyset, \emptyset)$, y el objetivo es pasar por al menos otro vértice y llegar a uno de los vértices terminales con el costo mínimo. Claramente, estos vértices terminales son pares ordenados cuyo término no vacío es un palíndromo y también el vértice $(\emptyset, \emptyset)$. Esto puede ser calculado fácilmente con el algoritmo de Dijkstra.

Con una buena implementación del algoritmo de Dijkstra, la complejidad por caso de prueba es $O(|V(G)| \log |V(G)| + |E(G)|) = O(ns(\log(ns) + n))$, dejando una complejidad total de $O(tns(n + \log s))$.

Implementación en C++:

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long int ll;
const ll oo = 1ll<<60ll;

int main(){
  #ifdef WozMit
  clock_t _start = clock();
  #endif
  int te; cin >> te;
  while (te--) {
    int n; cin >> n;
    vector<string> a(n);
    vector<int> c(n);
    for(auto &e: c) cin >> e;
    for (int i = 0; i < n; ++i)
```

```
    cin >> a[i];
  int nn = 0;

  // Identificar todos los vtxs y los terminales
  map<pair<string, string>, int> track;
  vector<int> terminals;
  track[{"", ""}] = nn++;
  for (auto &p: a){
    for (int i = 0; i <= (int)p.size(); ++i){
      if(i > 0){
        string t = p.substr(0, i);
        track[{"", t}] = nn++;
        bool is_pal = true;
        for (int j = 0; is_pal && 2*j < (int)t.size(); j++)
          if(t[j] != t[t.size() - 1 - j]) is_pal = false;
        if(is_pal) terminals.push_back(nn - 1);
      }
      if(i < (int)p.size()){
        string t = p.substr(i);
        track[{t, ""}] = nn++;
        bool is_pal = true;
        for (int j = 0; is_pal && 2*j < (int)t.size(); j++)
          if(t[j] != t[t.size() - 1 - j]) is_pal = false;
        if(is_pal) terminals.push_back(nn - 1);
      }
    }
  }

  // Construir el grafo (aristas)
  vector<vector<pair<int, int>>> G(nn);
  for(auto [e, x]: track){
    string t = e.first, s = e.second;
    for (int i = 0; i < n; ++i) {
      bool poss_left = true;
      // Verificar si es posible poner a[i] a la izquierda
      int mini = min(a[i].size(), s.size());
      for (int j = 0; poss_left && j < mini; j++)
        if(a[i][j] != s[s.size() - 1 - j]) poss_left = false;
      if(t != "") poss_left = false;

      bool poss_right = true;
      // Verificar si es posible poner a[i] a la derecha
      mini = min(a[i].size(), t.size());
      for(int j = 0; poss_right && j < mini; j++)
        if(t[j] != a[i][a[i].size() - 1 - j]) poss_right = false;
      if(s != "") poss_right = false;

      if(poss_left){
        if(a[i].size() >= s.size()){
          // Sobra a la izquierda
          pair<string, string> w = {a[i].substr(s.size()), ""};
          int a = x, b = track[w];
```

```
            G[a].push_back({b, c[i]});
          } else{
            // Sobra a la derecha
            pair<string, string> w = {"", s.substr(0, s.size() - a[i].
  size())};
            int a = x, b = track[w];
            G[a].push_back({b, c[i]});
          }
        }
        if(poss_right){
          if(a[i].size() >= t.size()){
            // Sobra a la derecha
            pair<string, string> w = {"", a[i].substr(0, a[i].size() - t
  .size())};
            int a = x, b = track[w];
            G[a].push_back({b, c[i]});
          } else {
            // Sobra a la izquierda
            pair<string, string> w = {t.substr(a[i].size()), ""};
            int a = x, b = track[w];
            G[a].push_back({b, c[i]});
    }
  }
      }
    }

    // Ejecutar el algoritmo de Dijkstra's desde 0
    vector<ll> d(nn, oo);
    d[0] = 0;
    priority_queue<pair<ll, int>> q;
    q.push({0, 0});
    while ((int)q.size() > 0) {
      int v = q.top().second;
      ll dv = -q.top().first;
      q.pop();
      // Evitar repetir estados innecesarios
      if(dv != d[v]) continue;
      for(auto edge: G[v]){
        int u = edge.first, w = edge.second;
        if(d[v] + (ll)w < d[u]){
          d[u] = d[v] + (ll)w;
          q.push({-d[u], u});
        }
      }
      // Ya salimos de 0, ya podemos ponerlo en su lugar
      if(v == 0 && d[0] == 0) d[0] = oo;
    }

    ll gg = d[0];
    for (auto e: terminals)
      if(d[e] < gg)
        gg = d[e];
```

```
119     if(gg == oo) cout << "Imposible\n";
120     else cout << gg << "\n";
121   }
122   return 0;
123 }
```