INTRODUCCIÓN A LOS CONCURSOS DE PROGRAMACIÓN

Dirigido a estudiantes de universidad, bachillerato y formación profesional

Autor:

SERGIO GARCÍA BAREA

Créditos

Autor: Sergio García Barea

Licencia

DEDICATORIA

A mi familia por apoyarme en todo momento, especialmente mi pareja, con una paciencia infinita.

A Miguel Ramos, antiguo profesor, mentor en los concursos de programación y actualmente compañero de profesión quien inició en mi y en otros compañeros la semilla que nos llevó al fascinante mundo de los concursos de programación, estimulándonos para competir.

A Jon Ander Gómez, antiguo profesor y mentor en los concursos de programación, quien continuó la labor de enseñarnos y estimularnos para competir.

A mis compañeros y rivales en las competiciones de programación, Mario Rodríguez, Ángel García, Leandro Gil, Marcos Calvo, Joan Pastor, Mercedes García, Iván Benito y muy en especial a Ximo Planells, compañero y competidor desde los inicios.

Índice

CAPÍTULO I: Requisitos para disfrutar este libro

¿Qué es la programación?

Según la Wikipedia, programación es "el proceso de diseñar, codificar, depurar y mantener el código fuente de programas computacionales. El código fuente es escrito en un lenguaje de programación".

A efectos prácticos, programar es dar instrucciones a un ordenador (la parte de codificar). Los ordenadores no tienen consciencia ni son inteligentes, así que **esas órdenes deben ser muy precisas y no ambiguas,** sino el ordenador no será capaz de realizarlas.

El que sean precisas, hace que deban estar muy meditadas antes (de ahí la parte de diseñar) y muy probadas después de codificarlas (la parte de depurar y mantener).

Estas instrucciones se proporcionan mediante los **lenguajes de programación.**

¿Qué es un lenguaje de programación?

Según la wikipedia "Un **lenguaje de programación** es un idioma artificial diseñado para expresar procesos que pueden ser llevadas a cabo por máquinas como las computadoras"

El fichero de texto escrito en un lenguaje, se llama **código fuente** y contiene las instrucciones del programa.

Antes de programar, hay que elegir que lenguaje de programación utilizaremos. Existen muchos: C, C++, Java, Python, Pascal, etc… Cada uno tiene sus ventajas e inconvenientes (portabilidad, legibilidad, velocidad, bibliotecas disponibles etc.)

A lo largo de este libro utilizaremos C/C++.

¿Cómo hago que un código fuente se ejecute?

Antes que nada hay que hacer una distinción: lenguajes interpretados, compilados y mixtos.

Los lenguajes interpretados, van leyendo línea a línea el código fuente e interpretan la instrucción que lea en cada momento. Requieren de un programa llamado intérprete. Python es un lenguaje interpretado.

Los lenguajes compilados, lo que hacen es previamente transformar un código fuente a programa ejecutable (En Windows generalmente extensión .exe). Esta transformación la realiza un programa llamado "Compilador". C/C++ son lenguajes compilados.

Los mixtos, son los lenguajes que se realiza una compilación del programa, pero esta no es un ejecutable real, sino que es interpretado por un interprete (Caso de la máquina virtual de java).

¿Qué debo saber antes de empezar con este libro?

Este libro intenta orientar al lector en el mundo de las competiciones de programación**. Pese a que resuelve problemas básicos y paso a paso, requiere unos conocimientos mínimos de programación**. Aprender a programar es un campo muy extenso y no podemos abordarlos desde cero en el libro.

Durante el libro, se explicarán algunos conceptos avanzados de programación, pero los conocimientos básicos deben ser adquiridos por el lector previa o paralelamente a la lectura del libro.

Recomendamos se aprenda C/C++ ya que será el lenguaje utilizado a lo largo del libro.

¿Es difícil aprender a programar?

Mucha gente dice: programar es difícil, es imposible, tienes que dedicar tu vida a ello, solo los genios pueden y así podría seguir... **Discrepo totalmente con estas opiniones**.

Al principio puede parecer más difícil de lo que es realmente. Como todo en la vida, es dedicarle tiempo y pasión. Ayuda mucho tener una motivación para aprender/mejorar y las competiciones de programación pueden ayudar en este aspecto.

¿Dónde puedo aprender a programar / mejorar mis habilidades?

Hay muchos libros y lugares en Internet donde podrás aprender a programar. A continuación, algunas **direcciones Web de interés** para comenzar:

- http://www.udacity.com
- http://coursera.org
- http://c.conclase.net

- http://hispabyte.net
- http://www.elrincondelc.com
- http://es.wikibooks.org
- http://www.cprogramming.com

¿Qué entornos de programación serán utilizados en el libro?

Lo explicado en este libro debería ser válido para cualquier compilador de C/C++ sin importar el editor que uses.

Pese a ello y con el fin de facilitar el desarrollo de los problemas que se expondrán hay algunos entornos gratuitos que se pueden encontrar en la Web:

- Codeblocks (Linux / Windows)
- Dev C++ (Windows)
- Pelles C (Windows)

CAPÍTULO II: Introducción a los concursos de programación

¿Qué es un concurso de programación?

Un concurso de programación es una competición donde se decide un ganador/ganadores en base a sus capacidades para realizar uno o varios programas que cumplan unos requisitos.

Hay de distintos tipos (programa mas bonito, mas usable, etc...), pero desde este libro **nos centraremos en los concursos de programación que tienen que ver con la resolución de problemas de algorítmica**.

Estos problemas generalmente consistirán en que un programa reciba unos datos siguiendo un patrón, y deberá procesarlos para devolver unos resultados. Se considerará un programa correcto cuando esos datos sean correctos para todos los casos.

¿Quién puede participar en un concurso de programación?

Depende de las bases del concurso. Hay concursos para estudiantes ESO/Bachillerato, para estudiantes de FP, para estudiantes universitarios y para todo el mundo.

Motivos para participar en un concurso de programación

Hay cientos de motivos, pero voy a nombrar los más destacados :

- **Superación personal**: generalmente, los problemas de los concursos de programación suponen un reto. Superarlo siempre es gratificante.
- **Adquisición de destreza**: al resolver estos problemas, mejoran tus habilidades como programador y analista, mucho más de lo que imaginas.
- **Prestigio**: la participación y buenos resultados en concursos de este tipo, está bien valorado tanto socialmente como a nivel de curriculum.
- **Selección de personal**: muchas grandes empresas (Facebook, Google) y algunas no tan grandes, utilizan este tipo de concursos para contratar a programadores con talento.

Concursos de programación en España

Hay 2 competiciones principales a nivel español : la **Olimpiada Informática Española** y **Programa-ME.**

La **Olimpiada Informática Española** esta pensada para alumnos de ESO, Bachillerato y Ciclos formativos de grado medio. Esta competición es individual. Habitualmente se realiza una fase previa online y los primeros clasificados acuden a una final presencial. Es una competición individual.

Mas información en su Web : http://www.olimpiada-informatica.org/

Programa-ME esta pensada para alumnos de Ciclos formativos de grado medio y grado superior. Se compite por equipos de tres personas (aunque con un solo ordenador). Se realizan concursos clasificatorios regionales (realizados en distintas sedes colaboradoras) y los primeros clasificados pasan a la final presencial. Es una competición realizada en equipos de tres personas y un ordenador.

Mas información en su Web : http://www.programa-me.com/

Además hay muchas competiciones locales a distintos niveles.

A nivel de ESO, Bachillerato y Grado Medio se realizan la Olimpiada informática de Castilla la Mancha, Extremadura o Murcia. Suelen ser de carácter individual.

Enlaces :

Castilla la mancha : http://olimpiadasinformatica.uclm.es/inicio/

Extremadura : http://www.oiex.org/

Murcia : http://olimpiada.inf.um.es/

A nivel universitario, existen muchas competiciones nacionales. Estas suelen realizarse con el fin de que las universidades realicen la selección de participantes que les representaran en competiciones internacionales. Estas suelen realizarse en la Universidad Politécnica de Valencia, en la Universidad Politécnica de Cataluña, en la Universidad Autónoma de Madrid, Universidad de Valladolid, etc... Suelen ser individuales y clasificatorias para la SWERC (Que nombraremos mas adelante).

Enlaces :

UPV (Valencia) : http://users.dsic.upv.es/grupos/clocalprog/

UPC (Barcelona) : http://concurs.lsi.upc.edu/

UAM (Madrid) : http://ce.azc.uam.mx/profesores/franz/acm/

UVA (Valladolid) : http://online-judge.uva.es/local/

Concursos de programación internacionales

A nivel internacional, existen los siguientes concursos:

Nivel ESO, Bachillerato y Ciclos Formativos de grado medio:

IOI (International Olympiad in Informatics): http://ioinformatics.org/

Esta competición es individual.

A nivel universitario :

existen varias fases regionales, que clasifican para la competición mundial, la **ACM-ICPC** (International Collegiate Programming Contest). http://icpc.baylor.edu/

La fase local correspondiente al sur oeste de Europa, donde se incluye España se llama **SWERC** (South Western European Regional Contest) http://swerc.eu/

Los mejores equipos de la **SWERC** se clasifican para las finales ACM-ICPC

Tanto las fases locales (SWERC y equivalentes) como la final de la ACM-ICPC se compite en equipos de tres personas usando un solo ordenador.

A todos los niveles:

Grandes corporaciones como Google, Facebook o paginas dedicadas a ello como Top-Coder, organizan competiciones abiertas a todo el mundo y con suculentos premios. A veces, estas competiciones se utilizan como sistema de selección de persona. Suelen ser de carácter individual.

Enlaces :

Google Code Jam: http://code.google.com/codejam/

Facebook Hacker Cup: http://www.facebook.com/hackercup

Lenguajes de programación aceptados

Hay dos tipos de competiciones, las que limitan los lenguajes que puedes usar y las que no los limitan (puedes usar cualquiera).

- **Lenguajes no limitados** : normalmente estas competiciones te dejan usar el lenguaje que quieras. Su funcionamiento es el siguiente. Ellos te envían un juego de prueba y tienes un tiempo

para devolver el resultado en un fichero de texto. Dado que el programa se ejecutara en tu ordenador, podrás usar el lenguaje que quieras.

- **Lenguajes limitados** : el concurso limita a algunos lenguajes **(Normalmente estos suelen ser C/C++ y Java)** y lo que hacen es que tu envías tu código fuente, ellos lo compilan internamente y lo prueban con juegos de prueba (que tu no ves, son secretos).

La mayoría de concursos de programación presenciales suelen ser limitados (OIE, IOI, SWERC, ICPC), pero los internacionales no presenciales (Google Code Jam, Facebook Hacker Cup).

Cómo determinan los jueces si un problema es correcto

La forma más utilizada para determinar si un programa es correcto, es usar un o varios juegos de prueba y comparar el resultado. Si ambos coinciden, el resultado es correcto. Según la competición, se considera un programa correcto si todos los juegos de prueba son correctos, aunque en otras se suelen dar puntos por cada juego de pruebas correcto. Al diseñar

un programa debemos pensar SIEMPRE en que funcione para todos los casos posibles. Los problemas usualmente tienen un límite definido (Ejemplo : habrá como máximo 1000 números). En los juegos de prueba, ese limite os aseguro que estará , habrá un caso que ponga los mil números. Y además probarán los casos más difíciles posibles. Debéis diseñar vuestros programas para que puedan con todo.

Lugares en Internet para entrenar

Hay muchos sitios para entrenar en Internet, donde encontrareis problemas y podréis resolverlos y ver si son correctos o no mediante un juez automático. La clave para obtener nivel en estas competiciones es realizar muchos problemas, así que entrenar por Internet es clave.

Algunos sitios interesantes :

Juez olimpiada informática española : En castellano, problemas similares a los que aparecen en la OIE

http://olimpiada-informatica.org/

USACO : En inglés aunque con algunas traducciones en castellano. Sitio que originariamente era para preparar al equipo norteamericano de la IOI,

pero abierto a todo el mundo, con multitud de problemas, teoría previa sobre como resolverlos, soluciones bien explicadas. Ideal para aprender.
http://www.usaco.org/

Juez ACM UVA : sitio con miles de problemas, alojado en la universidad de Valladolid. En inglés, problemas de todos los niveles.
http://uva.onlinejudge.org/

Cursos online: En **http://www.udacity.com** o **http://www.coursera.org** podéis encontrar cursos relacionados con la algorítmica que os serán útiles para mejorar vuestras habilidades.

Dada la gran cantidad de problemas que hay en la UVA, hay algunas **herramientas para elegir que problemas de la UVA pueden ser más interesantes**.
http://uhunt.felix-halim.net/
Y para buscar por temática
http://www.uvatoolkit.com/

CAPÍTULO III: Empezando, resolución de problemas sencillos

Para comenzar en las competiciones de programación, deben de construirse unos **buenos cimientos** y nunca empezar la casa por el tejado. Por ello, resolveremos a modo de prueba varios problemas sencillos y de paso veremos los principales problemas comunes que nos pueden surgir.

Cálculo de la letra del DNI

El DNI (Documento Nacional de Identidad) es un número identificativo único que posee cada ciudadano español. Este número tiene una letra al final. Dicha letra no es al azar, esta obtenida de una fórmula dependiente del número. **La letra se obtiene de la siguiente forma**: usamos nuestro numero de DNI y le aplicamos la operación “Modulo 23” (El resto de dividir un numero por 23). Este tipo de operaciones garantizan que el resultado siempre será un número entre 0 y 22.

El número obtenido, se busca su correspondencia en esta tabla, y de ahí obtenemos la letra.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

No se utilizan las letras: I, Ñ, O, U

La I y la O se descartan para evitar confusiones con otros caracteres, como 1, l o 0

Veamos un ejemplo. Si tenemos el DNI 12345678

Lo dividimos por 23 y sacamos su resto (Operación modulo)

12345678 % 23 = 14

Sabiendo que el resto es 14, buscamos la equivalencia en la tabla y tenemos la letra “Z”. **Ahora podemos decir que el DNI 12345678 tiene como letra Z.**

Si todo ha sido correcto y esta entendido, hemos dado el primer y más importante paso: saber que nos piden y saber resolver un caso sencillo a mano.

Ahora, vamos a pasar a la acción, a lo que podría ser un enunciado en un concurso de programación. Estos han de leerse siempre de forma detenida, teniendo en cuenta límites, excepciones y ejemplos que nos proporcionan.

PROBLEMA: CALCULO DE DNI (1)

ENUNCIADO: Realiza un programa que reciba por la entrada estándar un número de DNI y devuelva por la salida estándar la letra del DNI.

Algoritmo de cálculo del DNI

Usamos nuestro numero de DNI y le aplicamos la operación "Modulo 23" (El número obtenido, se busca su correspondencia en esta tabla, y de ahí obtenemos la letra.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

No se utilizan las letras: I, Ñ, O, U

EJEMPLO DE ENTRADA: Un único número de DNI (Entero positivo de 8 cifras).

EJEMPLO DE SALIDA: Letra del DNI en mayúsculas.

Entrada	**Salida**
12345678	Z

Comentando el problema:

Para resolver este problema, debemos hacer un programa que **lea de la entrada estándar (usualmente el teclado) un número y que imprima en la salida estándar (usualmente la pantalla) el resultado**. ¿Por qué digo usualmente? Por que a veces por practicidad, se puede redirigir la entrada o la salida desde un fichero. Esto significa que si se redirige la entrada, se simulara que se el contenido del fichero ha sido tecleado o en el caso de la salida, lo que se mostraría por pantalla se guardaría en un fichero.

Ejemplo:

Miprograma.exe < ficheroentrada.txt

En este programa "simulara" que se escribe por teclado el contenido del fichero "ficheroentrada.txt".

Miprograma.exe > ficherosalida.txt

En este programa se guardara en un fichero "ficherosalida.txt" lo que se habría mostrado por pantalla.

Miprograma.exe < ficheroentrada.txt > ficherosalida.txt

En este programa se simulara" que se escribe por teclado el contenido del fichero "ficheroentrada.txt" y además se guardara en un fichero "ficherosalida.txt" lo que se habría mostrado por pantalla.. Aclarado este punto (Os invito a que hagáis pruebas una vez tengáis realizado el programa) , hago una propuesta de solución al programa.

Código fuente:

```
#include <iostream>

using namespace std;

// Hasta aquí librerías necesarias para que funcione C++

int main(){
    // Correspondencias letra/numero
    char tabla[] = "TRWAGMYFPDXBNJZSQVHLCKE";
    // Declaramos la variable donde guardaremos datos
    int numeroDNI, numerom23;
    // Declaramos variable donde guardaremos la letra
    char letra;
    // Leemos de entrada estándar el numero del DNI
    cin >> numeroDNI;
    // Obtenemos el resto de dividir por 23
    numerom23=numeroDNI%23;
    // Guardamos el contenido de la posición en letra
    letra=tabla[numerom23];
    // Imprimimos letra y un salto de línea final
    cout << letra << endl;
    return 0;
}
```

Aquí los pasos son sencillos, leemos el dato, aplicamos los cálculos e imprimimos el resultado. **Deberíamos probarlo con mas DNIs** (se nos da uno de ejemplo, pero a modo de asegurarnos si el programa es correcto se antoja insuficiente. Hay que generar lo que se llama "Juegos de prueba").

Una vez esta todo correcto y funciona, hemos solucionado nuestro primer problema. Esto es algo bastante más complicado de lo que parece, y si lo has logrado, me tomo la libertad de felicitarte.

Aún así, **un mismo problema puede complicarse más**. En este caso, vamos a ver una nueva versión de este problema, pero que pueda tratar con más de un DNI a la vez. ¡¡ Que no cunda el pánico!! Con unos sencillos cambios podremos adaptarlo.

PROBLEMA: CALCULO DE DNI (2)

ENUNCIADO: Realiza un programa que reciba por entrada estándar una cantidad de DNIs, seguido de los números de DNI y devuelva por la salida estándar las letras de los DNI asociados. Habrá un máximo de 200 DNIs

Algoritmo de cálculo del DNI

Usamos nuestro numero de DNI y le aplicamos la operación "Modulo 23" (El número obtenido, se busca su correspondencia en esta tabla, y de ahí obtenemos la letra.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

No se utilizan las letras: I, Ñ, O, U

EJEMPLO DE ENTRADA: Cantidad de DNIs seguido de los números correspondientes, cada uno en una línea

EJEMPLO DE SALIDA: Letras asociadas a los DNIs, en mayúsculas Cada una por línea

Entrada	**Salida**
2 12345678 23456789	Z D

Comentando el problema:

Aquí la cosa ya se complica algo más. Aquí **deberemos leer primero la cantidad de números que hay.** Sabiendo esta cantidad, trataremos cada uno de forma independiente.

Y luego con un bucle (con tantas iteraciones como números hayan), calculara e imprimirá las letras de cada DNI.

El algoritmo para calcular las letras, es el mismo, reutilizaremos el anterior. ¿Para que hacer la misma faena dos veces? ;)

Código fuente:

```
#include <iostream>
using namespace std;
// Hasta aquí librerías necesarias para que funcione C++
int main()
{
    // Variable auxiliar
    int i;
    // Cantidad de DNIs
    int cantidad;
    // Correspondencias letra/numero
    char tabla[] = "TRWAGMYFPDXBNJZSQVHLCKE";
    // Declaramos la variable donde guardaremos el DNI
    int numeroDNI;
   // Declaramos la variable donde guardaremos el DNI%23
    int numerom23;
    // Declaramos variable donde guardaremos la letra
    char letra;
    // Leeremos la cantidad de DNIs
    cin >> cantidad;
// Bucle que se ejecutará tantas veces como la cantidad de
DNIs
    for(i=0;i<cantidad;i++){
        // Leemos de entrada estándar el numero del DNI
        cin >> numeroDNI;
        // Obtenemos el resto de dividir por 23
        numerom23=numeroDNI%23;
       // Guardamos el contenido de la posición en letra
        letra=tabla[numerom23];
        // Imprimimos letra y un salto de línea final
        cout << letra << endl;
    }

    return 0;
}
```

Si hemos llegado hasta aquí con todo correcto, enhorabuena de nuevo, hemos superado una nueva dificultad, que era hacer un programa que funcionara para varios DNIs.

Es habitual que nos den el límite de cuantos casos de prueba habrá dentro del problema. En este caso, hay 200 DNIs, así que al tomar nuestras decisiones deberemos tener en cuenta ese máximo **(RECORDAD : si hay un máximo, siempre habrá un juego de prueba que pruebe ese límite).**

En el caso concreto del DNI, este límite no es importante, pero en otros problemas, se dará el caso de que se podrán hacer dos programas completamente válidos, **uno más sencillo de codificar pero mas lento computacionalmente hablando** (necesite más operaciones), **y otro más rápido pero más complejo de codificar**. Si el límite es muy alto, es posible que lento pero sencillo no sea suficiente. Sin embargo, si el limite es pequeño, ¿para que complicarnos y perder tiempo haciendo el complejo?. **Esta decisión la tomaremos basándonos en los límites que se nos presenten**.

PROBLEMA : 3n + 1 (1)

ENUNCIADO: Considerando el siguiente algoritmo

```
While n!=1
        Print N
        If(n es par)
                N=N/2
        Else
                N=3*n+1
Print N
```

Si introducimos el número 22, se imprimirá la siguiente secuencia : 22 , 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8 , 4, 2, 1. Este algoritmo pertenece a la conjetura de Collatz. (Supuestamente todo número converge a 1, aunque no esta demostrado que dicha conjetura sea verdadera para todos los números existentes)

Realiza un programa que indique el tamaño del ciclo (cuantos números distintos tiene la secuencia obtenida de aplicar este algoritmo a un número dado). Habrá un máximo de 100 casos y todos serán mayores que cero y menores que un millón.

EJEMPLO DE ENTRADA: Un entero con cuantos juegos de prueba habrá, seguido de los números a calcular (uno por línea).

EJEMPLO DE SALIDA: cantidad de números distintos

Entrada	Salida
4 22 1 8 7	16 1 4 17

Comentando el problema:

En este problema básicamente **deberemos aplicar el algoritmo** (Si es par, dividir por 2 y si es impar, multiplicar por 3 y al resultado sumarle 1) y contar cuantos números aparecen hasta llegar al 1. Con los límites que nos dan, haciendo el algoritmo mas sencillo, nuestro problema funcionaria. (Estos son 100 casos y números entre 1 y un millón).

Recordad no hay que hacer ninguna comprobación de si la función converge o no, ya que se supone que siempre converge. En otros problemas parecidos, es posible que sea necesario hacer esa comprobación.

Código fuente:

```
#include <iostream>
using namespace std;

int main()
{
    int contador;
    int cantidad;
    int numero;
    cin >> cantidad; // Leemos cuantos números
    for (int i=0;i<cantidad;i++){ // Ejecutamos el bucle tantas
veces como números
        contador=1;// Ponemos el contador inicialmente a 1
        cin >> numero; // Leemos el numero
        while(numero!=1){ // Mientras sea distinto de 1, algoritmo
y contamos
            contador++;// Sumamos 1 al contador
            if(numero%2==0){
                numero=numero/2;
            }
            else{
                numero=(numero*3)+1;
            }
        }
        cout << contador << endl;// Imprimimos contador seguido de
un salto
    }
    return 0;
}
```

Esta versión ha sido sencilla y directa. Aun así, veremos otra versión del problema que **complicara esta situación y requerirá un algoritmo mas avanzado.**

Antes de plantar el problema pediría reflexionarais e investigarais por Internet sobre dos cuestiones :

1) **¿Podemos ahorrar tiempo de computo si guardamos resultados previos?** (Ejemplo, si sabemos que 8 son 4 números y calculamos 32, una vez lleguemos a 8 podríamos parar de calcular) .
2) **Sabiendo que los enteros suelen representarse internamente en la memoria del computador en Complemento A2 ¿Hay alguna forma más rápida de dividir?** ¿Os suena el operador desplazamiento de bits en C/C++?.

PROBLEMA: 3n + 1 (2)

ENUNCIADO: Considerando el siguiente algoritmo

```
While n!=1
        Print N
        If(n es par)
                N=N/2
        Else
                N=3*n+1
Print N
```

Si introducimos el número 22, se imprimirá la siguiente secuencia : 22 , 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8 , 4, 2, 1. Este algoritmo pertenece a la conjetura de Collatz. (Supuestamente todo número converge a 1, aunque no esta demostrado que dicha conjetura sea verdadera para todos los números existentes)

Realiza un programa que reciba dos números i y j (en ningún orden en particular) y nos diga entre esos números, ambos inclusive, el número de operaciones que tiene la secuencia mas larga. Los números serán mayores que 0 y menores que cien mil.

EJEMPLO DE ENTRADA: Varios pares de números i y j sin ningún orden en particular

EJEMPLO DE SALIDA: el ciclo mas largo entre el intervalo i,j propuesto

Entrada	Salida
1 20	20
30 80	116
101 110	114
1000 900	174

Comentando el problema:

En esta versión la cosa se complica más. No nos dicen cuantos casos de prueba habrá como máximo. Esto es un caso inusual, **lo pongo como ejemplo de cosas que pueden ocurrir**, y si os pasa en un concurso, **lo lógico seria pedir una clarificación de ese dato que falta**.

Como no sabemos la cantidad y en además hay que hacer mas comprobaciones (ahora **se comprueban rangos, no números sueltos**), vamos a suponer que debemos hacer la **versión mas óptima posible del programa**.

Antes de comenzar, debemos fijarnos que dice **que los números i y j pueden estar en cualquier orden**, por lo cual no implica que uno sea menor que el otro, así que esto deberemos **comprobarlo en el código. Ojo con estas trampas,** aquí lo he puesto claramente en un juego de prueba de ejemplo, pero puede ser que en el enunciado lo ponga, pero los casos de prueba de ejemplo sigan un orden (por ejemplo, el primero

menor que el segundo) y **una cosa tan sencilla, nos despiste y perdamos un tiempo valioso** en averiguar como resolverlo.

En este caso lo solucionaremos tomando que el primero es el menor, y en el caso de que no sea así, **invertiremos los valores** (metemos en i el valor de j y en j el valor de i).

Para resolver este ejercicio de la forma más óptima posible, primero tendremos que considerar la posibilidad de **usar memoria para ahorrar cálculos**.

Dado que solo hay cien mil números a probar, podemos crear un vector de enteros de un millón de posiciones e inicializar sus posiciones a 0. En una posición que haya un **cero significará que ese valor aún no se ha calculado** (recordamos que en este problema no hay ningún caso que de 0 ciclos. Si en otro problema lo hubiera, podríamos usar otro valor no posible, como -1, un número muy grande,).

Distinto de cero, significa que la posición en concreto, tiene ese número de ciclos (Por ejemplo, la posición 22 guardaría un 16). Así calcularemos

el millón de números antes de la ejecución y no tendremos que calcularlo bajo demanda (con lo que se podrían repetir cálculos).

Además, **no es necesario calcular completamente el millón de números**. Cuando el algoritmo nos lleve a un número, si ya esta calculado, simplemente deberemos sumarle los pasos que llevemos a los que le queden a ese número y parar de calcular.

Ejemplo: Si calculamos el 8, su secuencia será 8,4,2,1, dando un resultado de 4. Si calculamos el 16, su secuencia seria 16,8,4,2,1. Fijaros que a partir del 8 es la misma, así que seria sumarle los pasos que lleve al número de pasos del 8, y si el 8 ya estaba calculado, ahí pararía y evitaría cálculos.

Puede existir el problema que por límites del entorno de programación o del juez, no podamos usar tanta memoria (un millón de enteros, son 4 millones de bytes, es aceptable pero algunos entornos pueden ser quisquillosos). Además también es posible que **en cálculos internos, se usen números superiores al millón**. Esto lo hemos tenido en cuenta en el código fuente, con una constante que define hasta que valores se pueden guardar en memoria.

Hay otra posible optimización, esta dependiente del lenguaje. En C/C++, hay una forma de dividir mas rápida que es usando el operador de desplazamiento de bits a la derecha (a la izquierda sirve igual para multiplicar por 2). Esto tiene que ver con las propiedades de las potencias y la representación de enteros en Complemento A2.

Haciendo x=x>>1; desplazamos un bit a la derecha, lo que equivale a dividir por dos. Asimismo x=x<<1; desplaza un bit a la izquierda, lo que equivale a multiplicar por 2.

Aquí una posible solución con precálculos solo hasta 100000.

Código fuente:

```
#include <iostream>
using namespace std;
#define MAXTAM 100000
int resultados[MAXTAM];// Vector de 0 a 99999

int main()
{
    int i,j;
    // Inicio del pre-calculo
    for (int x=0;x<MAXTAM;x++){
        resultados[x]=0; // Inicializamos a cero (no usado)
    }
    resultados[1]=1; // Caso base, ya sabemos el resultado
    // Ejecutamos el bucle tantas veces como números

    for (int x=2;x<MAXTAM;x++){
        int ciclos=0;
            // Para llevar la cuenta de ciclos en cada numero
        int num=x;// Auxiliar para operar con el numero actual
        while(num!=1) // Mientras no converja
        {
            ciclos++;
            if(num%2==0){
                num=num>>1;// equivalente a num=num/2;
            }
            else{
                num=(num*3)+1;
            }
            /* Aquí solo operamos si el resultado esta en el rango
           de posibles guardados y además esta guardado
            Eso significa que ya tenemos solución, así que
           romperemos el bucle */
            if(num<MAXTAM && resultados[num]!=0){
                break;
            }
        }
           /* Asignamos el ultimo numero encontrado (en el peor
           caso será el 1) mas los ciclos acumulados como resultado
           */
        resultados[x]=ciclos+resultados[num];
    }
    // Fin del pre-calculo

    /* Con este tipo de línea como no sabemos cuantos pares i, j
habrá Leeremos i y j mientras haya datos de entrada */
    while(cin >>i >> j){
        if(i>j) // Caso de i mayor que j, algoritmo de intercambio
```

```
        {
            int aux=i;
            i=j;
            j=aux;
        }
        int máximo=0; // máximo inicial de ciclos en un rango
        for (int x=i;x<=j;x++){ // Buscamos en el rango un ciclo
mayor
            if(resultados[x]>máximo){
                máximo=resultados[x];
            }
        }
        cout << máximo << endl; // Imprimimos el mayor ciclo
encontrado
    }
    return 0;
}
```

CAPÍTULO IV: Problemas sin estrategia concreta

Los problemas sin estrategia concreta son problemas que no tienen un algoritmo "famoso" asociado a su resolución. Entonces... ¿Cómo debemos afrontarlos?

La idea principal es leer el enunciado, tener claros los límites de los datos de entrada y hacer lo que pide.

Algunos de estos problemas son sencillos, porque con hacer lo que se pide, funcionarán a la perfección. Otros, para que funcione perfectamente, requerirán algún "Truco" o propiedad matemática que ayude en los cálculos, con lo que dificultará su diseño.

Tipo de problemas sin estrategia concreta

Hay muchísimos tipos de problemas sin estrategia concreta , aunque habitualmente son estos : Simulación de un algoritmo o proceso, (3n+1, simular un procesador, etc...) o matemáticos (decidir si un número es primo).

A continuación plantearemos algunos problemas

PROBLEMA: Simulador CPU

ENUNCIADO: Una determinada CPU tiene 5 registros (numerados del 0 al 4) y una memoria RAM de 100 palabras, empezando por la cero. Cada registro o palabra, contiene un entero entre 0 y 999. Las instrucciones se guardan en RAM y son las siguientes

100 Parar
2XY Inicializar el registro X con el valor Y
3XY poner el valor del registro X en el registro Y
4XY Sumar al registro X el valor Y
5XY multiplica el registro X por el valor Y
6XY inicializar la dirección RAM en el registro Y con el valor del registro X

EJEMPLO DE ENTRADA: Las palabras de RAM (hasta un máximo de 100) , cada una en una línea. El programa comenzará su ejecución a partir de la primera palabra. Las palabras no especificadas deberán tener la instrucción parar (100). **El programa no tendrá instrucciones inválidas y siempre llegará a una instrucción parar.**

EJEMPLO DE SALIDA: El número de instrucciones ejecutadas(parar incluida)

Entrada	Salida
239 431 532 535 217 314 634 222 432	8

Comentando el problema:

Para afrontar este problema, hay que conocer como funciona un procesador. **El procesador lee una instrucción(palabra), la ejecuta y continúa en la siguiente**. En un procesador real, es algo mas complicado, pero aquí en este procesador simple, seguiremos esa línea. ¿Como resolveremos el problema? **Leemos la instrucción apropiada, si es parar, paramos e imprimimos el resultado. Si es otra, hacemos lo que diga la instrucción y continuamos en la siguiente.**

Hay que fijarse, que **algunas instrucciones modifican la RAM**. Eso puede llevar a que modifiquen una instrucción ya existente. Por ejemplo, en el caso de prueba propuesto, modifican la palabra 7 y meten un valor 100 (instrucción parar), por eso se para a las 8 instrucciones.

Otra cuestión a tener en cuentas es si nos merece la pena **tratar las palabras como cadenas de longitud 3 o como números enteros**. Tratarlos como cadenas podría darnos ciertas ventajas (Por ejemplo, separar el código de instrucción). El tratarlos como números, nos daría la ventaja de que seria más fácil operar con ellos.

Código fuente:

```
#include <iostream>
#include <cstring> // Para usar funciones de cadena
using namespace std;
// Función que convierte un carácter en entero
int charToInt(char c){
    return c-'0';
}

int main(){
    int registros[5];
    char palabras[100][4]; // 100 palabras de 3 caracteres
    char paltmp[4];
    int posactual=0;
    int i;
    for(i=0;i<100;i++){
        strcpy(palabras[i],"100");// Inicializamos todas a 100
    }
    // Para leer palabras hasta que se acabe el fichero
    while(cin >> paltmp){
        strcpy(palabras[posactual],paltmp); // Copia paltmp en la
posición actual
        posactual++; // Posición siguiente
    }
    posactual=0; // La posición actual vuelve a 0
    do{
        // Extraemos de la palabra los 3 enteros
        int código=charToInt(palabras[posactual][0]);
        int operandoX=charToInt(palabras[posactual][1]);
        int operandoY=charToInt(palabras[posactual][2]);
        if(código==2) { // Operación inicializar registro
            registros[operandoX]=operandoY;
        }
        else if(código==3){// Operación valor de un registro X a Y
            registros[operandoY]=registros[operandoX];
        }
        else if(código==4){// Operación suma al registro
            registros[operandoX]+=operandoY;
        }
        else if(código==5){// Operación suma al registro
            registros[operandoX]*=operandoY;
        }
        else if(código==6){// Operación valor de un registro a otro
            sprintf(paltmp,"%d",registros[operandoX]); // Entero a
cadena
            strcpy(palabras[registros[operandoY]],paltmp);
        }
        posactual++; // Incrementamos la posición actual
```

```
    }while(strcmp(palabras[posactual],"100")!=0);   // Mientras la
palabra actual no sea 100
    cout << posactual+1 << endl; // +1 para contar la instrucción 100
    return 0;
}
```

En la solución se ha utilizado cadenas de caracteres para representar las palabras, enteros para representar los registros y al extraer los operandos, se ha transformado cada uno de cadena a entero.

Se han utilizado algunas **funciones de cadenas** de C (para ello se ha incluido la librería cstring).

Estas son :

- **strcmp(cad1,cad2) :** devuelve cero si las cadenas son iguales, distinto de cero en caso contrario.
- **strcpy(destino, origen) :** copia la cadena origen en la cadena destino (ojo, en las cadenas en C/C++ no basta con usar el operador =.
- **sprintf(cadena,"%d",entero) :** la potente orden sprintf, que convierte un patrón (en este caso "%d" es un entero) en cadena.

Para otros problemas, os pueden ser útiles otras funciones de cadenas:

- **atoi(cadena) :** convierte una cadena a entero.
- **strcat(destino, origen) :** concatena el origen al final del destino.
- **strlen(cadena) :** devuelve un entero con el tamaño de la cadena

- **strlwr(cadena):** Transforma una cadena todo a minúsculas

Para convertir un solo carácter a entero, se ha usado una función propia llamada charToInt. Esta recibe un carácter, y lo trata como su código ASCII (un numero que representa el carácter). En el código ASCII hay una propiedad que el carácter numérico que sea, menos el carácter '0', da su numero en entero (solo valido para números 0 al 9).

Sabiendo esto todo esto, la complejidad del problema reside en seguir los pasos, hacer todo tal cual, sin ninguna trampa extraña y en fijarse muy bien y comprobar que los pasos se han seguido correctamente.

Este ejercicio podría complicarse con más elementos (más instrucciones, posibilidad de saltos de una instrucción a otra), pero esos elementos solo obligarían a fijarnos más y realizar más pruebas, no cambiara la estrategia general a seguir.

PROBLEMA: ¿Es primo?

ENUNCIADO: Realiza un programa que lea una lista de números y diga si son primos o no. Se entiende que un número es primo si es divisible únicamente por si mismo y por la unidad y se entiende por divisible que el resultado de su división es entero (o dicho de otro modo, que su resto es cero). En el ejercicio se tendrá en cuenta que el número 1 no es primo

EJEMPLO DE ENTRADA: Un número por línea entre 1 y 10 millones ambos inclusive,

EJEMPLO DE SALIDA: imprimirá SI sí es primo, NO si no lo es

Entrada	Salida
1	NO
2	SI
4	NO
7	SI
11	SI
210	NO

Comentando el problema:

Para resolver este problema, deberemos aplicar matemáticas.

Primero tener en cuenta que el **concepto divisible es que el resto de la división sea cero.** Podemos usar el **operador módulo** (**%** en c) para saber si un número es divisible o no por otro.

Segundo, sabemos que todo número es divisible por si mismo y por la unidad, así que para saber que un número es primo, deberemos demostrar que los números que nuestro numero no es divisible por los números que hay "en medio". Ejemplo : para comprobar que 7 no es primo, debemos comprobar que 7 no es divisible por 2,3,4,5 y 6. Si es divisible por alguno, no es primo. Si no lo es por ninguno, es primo (en este caso 7 es primo).

Después de saber esto, es importante este rango de números se puede reducir por una propiedad matemática (y de paso reducir el tiempo de ejecución del problema).

Solo hace falta comenzar a **contar desde la raíz de nuestro número.** Por ejemplo, en el caso de 22, la raíz cuadrada es 4.69, por lo cual deberíamos comprobar desde el 2 hasta el 4.

Por último y no menos importante, hay que fijarse en los límites. Una variable de tipo entero podemos usarla para representar el valor más alto (10 millones). Pero si el valor fuera otro mucho mayor, no bastaría, ya que la precisión de los enteros es limitada.

Para ello se pueden utilizar bibliotecas para usar números grandes.

En C/C++ no hay forma nativa de utilizar números grandes, aunque hay bibliotecas externas como **GMP http://gmplib.org/** que ayudan en esta tarea.

Lo de no tener bibliotecas nativas, os limita para las competiciones en las que enviéis el código fuente y se compile en el juez, pero no para aquellas que dejen usar cualquier lenguaje de programación.

Código fuente:

```
#include <iostream>
#include <cmath> // Librería para usar matemáticas
using namespace std;
// Función que devuelve 1 si es primo o 0 si no lo es
int esPrimo(int num){
    if(num==1){ // Caso de 1, no es primo
            return 0;
        }
        // Calculamos la raíz, y nos quedamos con la parte
    entera
        int raíz=(int)sqrt((double)num);
        for(int i=2;i<=raiz;i++){
            if(num%i==0){// Caso es divisible
                return 0; // Devolvemos no primo
            }
        }
        return 1; // Si ninguno ha sido divisible,
    devolvemos primo.
    }

    int main()
    {
        int num;
        while(cin >> num){ // Leemos todos los primos
            if(esPrimo(num)){
                cout << "SI"<< endl;
            }
            else{
                cout << "NO"<<endl;
            }
        }
    }
```

Apunte extra : en este ejercicio recibimos un número y comprobaremos si un es primo o no, pero hay un método para crear una lista de primos pre-calculada, **la criba de Eratóstenes** (Útil en otros ejercicios). aquí tenéis el algoritmo, donde en la lista **0 significa no primo y 1 significa primo**.

```
void criba(unsigned char m[], int tam){
    int i, h;

    m[0] = 0; // 0 y 1 no son primos
    m[1] = 0;
    for(i = 2; i <= tam; ++i) m[i] = 1; // Marcamos todos como primos

    for(i = 2; i*i <= tam; ++i) {
        if(m[i]) {
            for(h = 2; i*h <= tam; ++h)
                m[i*h] = 0; // Descartamos los múltiplos de los
primos
        }
    }
}
```

PROBLEMA: ¿Es palíndromo?

ENUNCIADO: Realiza un programa que lea una cadena de caracteres y diga si es un palíndromo o no, sin distinguir entre mayúsculas y minúsculas. Se considera palíndromo toda cadena que al revés se lee exactamente igual que al derecho.

EJEMPLO DE ENTRADA: Una cadena por línea, de cómo máximo 200 caracteres. Las cadenas no contendrá espacios.

EJEMPLO DE SALIDA: Imprimirá SI sí es palíndromo, NO si no lo es

Entrada	**Salida**
OTTO MoTTO ANa LInO	SI NO SI NO

Comentando el problema:

Para resolver este problema, deberemos darle la vuelta a la cadena y ver si son iguales. Esa es una posible solución (la más humana).

Pero es mas sencillo **aprovecharse de las herramientas de programación** y colocar una variable que apunte al principio de la cadena y otra al final. Si los valores del principio y final de la cadena son distintos, no es palíndromo. Si lo son, la variable del inicio se incrementa y la del final se decrementa, y volvemos a comprobar si son iguales. Si no lo son, no es palíndromo. Cuando hayamos acabado de comprobar todas (es decir, cuando la variable de inicio, sea ya mayor o igual a la del final), si todos los casos han sido iguales, es que la cadena es un palíndromo.

Código fuente:

```
#include <iostream>
#include <cstring>
using namespace std;
// Función que devuelve uno si es palíndromo 0 si no
int esPalindromo(char *cad){
    int tam=strlen(cad); // Saca el tamaño de la cadena
    int inicio=0, final =tam-1; // Establecemos inicio y final

    while(inicio<final){
        if(cad[inicio]!=cad[final]){ // Caso no palíndromo
            return 0;
        }
        inicio++;// Inicio hacia adelante
        final--;// Final hacia atrás
    }
    return 1;// Todos iguales, es palíndromo
}

int main(){
    char cadena[201];// Cadena de 200 caracteres
    while(cin >> cadena) // Leemos cadenas
    {
        strlwr(cadena); // Pasa toda la cadena a minúsculas
        if(esPalindromo(cadena)){
            cout << "SI" << endl;
        }
        else{
            cout << "NO" << endl;
        }
    }
    return 0;
}
```

En esta solución, leemos la cadena y **para evitar la diferencia entre mayúsculas y minúsculas, pasamos toda la cadena a minúsculas** y operamos así con ella. Después simplemente aplicamos la estrategia propuesta (apunte al inicio y al final y van desplazándose) y obtenemos la solución.

De cara al siguiente problema, que es bastante más complejo que los anteriores, quiero plantear dos cuestiones para reflexionar:

1) **¿Como se programaría un generador de palíndromos?** La pista es que una vez tengamos una cadena normal, la que sea, hacer un palíndromo es "añadirle" un "espejo".
2) Tomando únicamente números **¿Qué hay en mayor cantidad, primos o palíndromos?**
3) **¿Que cuesta más computacionalmente**, comprobar si un número es primo, comprobar si un número es palíndromo o generar un número palíndromo?

PROBLEMA: ¿Es primo y palíndromo?

ENUNCIADO: Realiza un programa que lea únicamente dos números ay b enteros entre 5 y 100.000.000, siendo a<=b. Estos serán un rango donde ambos estarán incluidos en el que habrá que comprobar que números son primos palíndromos e imprimirlos en orden de menor a mayor. Se considera primo palíndromo aquel número que es primo y palíndromo a la vez.

Por ejemplo 101 es palíndromo y además es primo.

EJEMPLO DE ENTRADA: Dos números a y b, siendo a<=b

EJEMPLO DE SALIDA: Listado de primos palíndromos en ese rango (ambos inclusive) impresos cada uno en una línea, de menor a mayor.

Entrada	**Salida**
5 500	5 7 11 101 131 151 181 191 313 353 373 383

Comentando el problema:

En este problema, **debemos fijarnos inicialmente en el rango**. Nos dicen que es entre 5 y 100.000.000 números, por lo cual podemos intuir que ese caso es el caso peor (el que requerirá mas cálculos) y será probado en los juegos de prueba.

La aproximación más sencilla al problema seria recorrer el rango, **ir probando si es primo y es palíndromo** a la vez con las funciones de los anteriores problemas, y si lo es lo imprimimos. Parece sencillo ¿No? Pues en este caso, lamento decir que **la cosa no será tan fácil, ya que el rango es MUY amplio** y deberemos realizar un análisis para tomar una solución más compleja pero más rápida.

Aquí hay que hacer un ejercicio de análisis:

1) Supongamos que queremos comprobar todos los números uno a uno. Habría que comprobar si es palíndromo y luego primo o al revés. Lo ideal seria primero hacer la operación menos costosa. **¿Que cuesta menos computacionalmente, saber si es primo o palíndromo?** En término medio, **cuesta menos saber si es palíndromo que si es primo**. Solo con la elección de este orden, el tiempo de ejecución

variaría bastante, pero no suficiente para que nuestro programa funcionara en tiempo razonable.

2) Para calcular primos rápidos, se podría usar **la criba de Eratóstenes** que mencione antes. ¿Seria útil en este caso? La respuesta es no, ya que necesitaríamos **un vector de 100.000.000 de posiciones, cosa que no es viable** por cuestiones de memoria.
3) Ahora otra cuestión **¿Que hay más? ¿palíndromos o primos?** La respuesta es fácil, con una lista de primos y otra de palíndromos, se observa que hay más palíndromos que primos. Esta cuestión es interesante para pensar en la siguiente, aunque no nos ha ayudado mucho.
4) **¿Que es mas rápido, comprobar todos los números si son palíndromos o generar palíndromos?** La segunda opción es bastante más rápida. Aunque hay muchos palíndromos, son pocos comparados con números normales. Si en vez de comprobarlos, los generamos, nos ahorramos muchos números a comprobar.
5) **¿Como generamos palíndromos?¿Y en orden?** La idea podría ser hacer una función que le pasaras un número y de ahí te generara primero todos los palíndromos pares con ese número y luego los impares. Por ejemplo, si le pasas 1, te generaría primero los impares

(11) y luego los impares (101,111,121,131,141,151,161,171,181,191). debería soportar palíndromos de hasta 8 caracteres de anchura. Son 8 y no 9, porque el caso de 9 cifras, que es 100.000.000, es solo un caso de anchura 9 y ni es primo ni palíndromo, así que podemos descartarlo.

Para generar los pares, haríamos efecto espejo (Generas un número y pones su reflejo) y **para los impares**, ese mismo reflejo, pero con los dígitos del 0 al 9 en medio.

Tras este análisis, la opción más sencilla esta clara : **generamos palíndromos en orden, y comprobamos si son primos o no**. Los que sean primos los imprimimos.

Código fuente:

```
#include <iostream>
#include <cmath>
using namespace std;
// Función que calcula si es primo o no
int esPrimo(int num){
    if(num==1){ // Caso de 1, no es primo
        return 0;
    }
    // Calculamos la raíz, y nos quedamos con la parte entera
    int raiz=(int)sqrt((double)num);
    for(int i=2;i<=raiz;i++){
        if(num%i==0){// Caso es divisible
            return 0; // Devolvemos no primo
        }
    }
    return 1; // Si ninguno ha sido divisible, devolvemos primo.
}
// Función que recibe numero, si genera par o impar y los limites
void generar(int n,int esImpar,int limini,int limfin){
    char buffer[16];
    int ini, fin;
    int numero;

    sprintf(buffer,"%d",n);// Pasamos el numero a cadena
    // Marcamos el final y el inicio desde donde haremos el efecto
espejo
    fin = strlen(buffer);
    ini = fin - 1;

    while(ini >=0) { // Mientras quede cadena
        buffer[fin]=buffer[ini]; // Copiamos el carácter
           ini--; // Hacia atrás
           fin++;// Hacia adelante
    }
    buffer[fin] = '\0'; // \0 final para el formato correcto de la
cadena
    // Caso impar, añadiremos del 0 al 9 en medio
    if(esImpar==1){
        int tam=strlen(buffer)-1;
        int medio=tam/2;
        // Desplazamos los de detrás del medio un hueco a la derecha
        for (int x=medio+1;x<=tam;x++){
            buffer[x+1]=buffer[x];
        }

        for (int x=0;x<=9;x++){
            // Truco para poner en medio el carácter correspondiente
```

```
al numero
             buffer[medio+1]='0'+x;
            numero = atoi(buffer); // Pasamos a entero
            // Si esta en el limite y es primo, imprimimos
             if(numero<=limfin && numero>=limini){
                 if(esPrimo(numero)){
                     cout << numero << endl;
                 }
             }
         }
     }
     else
     {
            numero = atoi(buffer); // Pasamos a entero
            // Si esta en el limite y es primo, imprimimos
         if(numero<=limfin && numero>=limini){
           if(esPrimo(numero)){
                  cout << numero << endl;
           }
         }
     }
}

int main(){
    int limini,limfin;
    int i;
    cin >> limini >> limfin;
    //Primero números del 5 al 9 directamente
    for (i = 5; i <= 9; i++)      {
        if(i<=limfin && i>=limini){ // Si están en los limites
          if(esPrimo(i)){
                 cout << i << endl;
           }
         }
     }
    for (i = 1; i <= 999; i++)    { //Calculamos pares e impares
            generar(i,0,limini,limfin); // Generamos pares
            generar(i,1,limini,limfin); // Generamos impares
    }
    for (i = 1000; i <= 9999; i++)       { // aquí solo hacen falta
pares
            generar(i,0,limini,limfin); // Generamos pares
    }
    return 0;
}
```

En la solución propuesta, **se calculan siempre todos los números** (Caso máximo), **pero solo se imprimirán los que estén en el rango**. No es la solución más elegante, pero es la más simple y que funciona (ya que **funciona en el caso peor siempre**).

Otra forma quizás mas sencilla (y por lo tanto, mas apropiada para el contexto de un concurso), es fijarse que solo puede haber palíndromos de hasta anchura 8. Al ser solo 8 casos, en vez de hacer una función de carácter general que es mas corta, pero mas compleja de pensar, podíais haber hecho 8 funciones distintas. Que sea menos elegante, no significa que sea menos apropiada ni menos valida.

Reutilización de código

Nótese que en la implementación hemos reutilizado la función esPrimo que hemos utilizado en el ejercicio anterior. Es una buena práctica el guardar código y saber cuando reutilizarlo.

También es una buena práctica hacerse "apuntes" con funciones mas utilizadas para poder reutilizarlas rápido en ejercicios parecidos. Las funciones esPrimo y esPalindromo, así como otras que veremos mas adelante, podrían ser candidatas a formar parte de esos apuntes.

PROBLEMA: Cambios de base

ENUNCIADO: Realiza un programa que lea 3 números. El primer número será la base actual de un número. el segundo, la base deseada y el tercero el número con el que operaremos. Los datos se proporcionarán de tal forma que el cambio de base siempre será posible.

OJO: recordad algunas bases números en determinadas bases llevan letras, como por ejemplo BC en hexadecimal (base 16). Las letras se expresarán en mayúsculas.

EJEMPLO DE ENTRADA: Tres números, base actual, base deseada y numero para operar.

EJEMPLO DE SALIDA: Listado de primos palíndromos en ese rango (ambos inclusive) impresos cada uno en una línea, de menor a mayor.

Entrada	**Salida**
2 10 100	4
16 10 A	10

Antes de resolver el problema, hay que conocer los **algoritmos de cambio de base**. Podéis buscar información en Internet para profundizar, pero en la solución tendréis dos funciones de cambio de base (valida hasta base 40) que podréis **reutilizar** a vuestro gusto en futuros problemas.

En estas funciones, una pasara **de cualquier base a decimal**, y otra pasara **de decimal a cualquier base**.

Si queremos pasar de una base distinta de decimal X a otra también distinta de decimal Y, deberemos **pasar a X a decimal** y el valor que obtengamos **en decimal, lo pasamos a la base Y**.

Aquí veremos la solución al problema.

Código fuente:

```
#include <iostream>
using namespace std;
// Convierte un numero de decimal a otra base
void fromDec(int n, int b, char *s) {

    char digitos[40] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    int p = 0;

    if (n == 0)
            s[p++] = '0';

    // Tomamos restos de dividir por la base
    while (n != 0) {
            s[p] = digitos[n%b];
            n /= b;
            p++;
    }

    // invertir la cadena
    for (int i = 0; i < p/2; i++) {
            int t = s[i];
            s[i] = s[p-i-1];
            s[p-i-1] = t;
    }
    s[p] = '\0';
}

// convierte un número a decimal
int toDec(char *s, int b) {
    int res = 0;
    int p = 0;
    // Aplicamos teorema fundamental de la numeración
    while (s[p] != '\0') {
            int digito;
            if (s[p] >= '0' && s[p] <= '9')
                    digito = s[p] - '0';
            else
                    digito = s[p] - 'A' + 10;

            res = res*b + digito;
            p++;
    }
    return res;
}

int main(){
    char resultado[100]; // Donde guardaremos la conversión
```

```
    char num[100]; // Recordad algunas bases pueden llevar letras
    int baseActual,baseDeseada;
    cin >>  baseActual>> baseDeseada;
    cin >> num;
    // Si se desea la misma base actual, se imprime tal cual
    if(baseActual==baseDeseada){
        cout << num << endl;
    }
    // Si esta en base 10, solo pasamos a la deseada
    else if(baseActual==10){
        int n=atoi(num);
        fromDec(n,baseDeseada, resultado);
        cout << resultado<<endl;
    }
    // Si se desea base 10, se pasa a esa base y se imprime
    else if(baseDeseada==10){
        int n=toDec(num, baseActual);
        cout << n << endl;
    }
    // Cambio entre bases distintas a 10, pasar a base 10
    // y de ahí a la base deseada
    else{
        int n=toDec(num, baseActual);
        fromDec(n,baseDeseada, resultado);
        cout << resultado<<endl;
    }

}
```

CAPÍTULO V: Problemas sobre cadenas de caracteres

El uso de cadenas de caracteres para solucionar parte de algunos problemas, es imprescindible. Ya las hemos usado en el capítulo anterior, pero repasaremos las funciones y técnicas más importantes del uso de cadenas.

Representación ASCII

En C/C++, los caracteres están representadas mediante código ASCII (American Standard Code for Information Interchange). Este código utiliza un byte por carácter y nos permite representar hasta 255 caracteres distintos. Dentro del ASCII hay variantes, y no todas utilizan los 255 caracteres. Es fácil encontrar tablas ASCII que muestren todos los caracteres en Internet.

Cada "numero" en binario es un carácter. Por ejemplo la letra 'A' es el número 65, la letra 'B' 66, la 'C' 67… la 'a' '97', la 'b' 98…

Asimismo hay caracteres especiales, como el salto de línea, el tabulador, etc…

Operando ASCII

En el código ASCII las letras mayúsculas, minúsculas y los números están seguidos. Esto nos permite:

- Si queremos hacer un bucle de ‘a’ a la ‘z’ por ejemplo, podemos usar esta propiedad.
- Para pasar un carácter de mayúsculas a minúsculas, usamos la siguiente formula : Carácter mayúsculas – ‘A’ + ‘a’ , por ejemplo si queremos pasar ‘C’, haremos ‘C’ - ‘A’ + ‘a’.
- Para minúsculas a mayúsculas, Carácter minúsculas – ‘a’ + ‘A’, por ejemplo si el carácter es ‘b’, haremos ‘b’ –‘a’ + ‘A’.
- Podemos pasar un número a su valor entero haciendo carácter número – ‘0’, por ejemplo ‘7’ seria ‘7’ – ‘0’

Cadenas de caracteres

En C/C++ las cadenas de caracteres se representan como un vector, en el que cada posición contiene un carácter ASCII y al final contiene el carácter nulo (‘\0’) para indicar que ha acabado la cadena. Para operar correctamente, es necesario que las cadenas cumplan con este formato.

Las funciones más importantes para operar con cadenas de caracteres son :

- **strcmp(cad1,cad2) :** devuelve cero si las cadenas son iguales, distinto de cero en caso contrario.
- **strcpy(destino, origen) :** copia la cadena origen en la cadena destino (ojo, en las cadenas en C/C++ no basta con usar el operador =.
- **strcat(destino, origen) :** concatena el origen al final del destino.
- **strlen(cadena) :** devuelve un entero con el tamaño de la cadena
- **strlwr(cadena):** Transforma una cadena entera a minúsculas
- **strupr(cadena):** Transforma una cadena entera a mayúsculas
- **sprintf(cadena,"%d",entero) :** la potente orden sprintf, que convierte un patrón (en este caso "%d" es un entero) en cadena.
- **atoi(cadena) :** convierte una cadena a entero.

A continuación veremos un problema de cadenas de caracteres sencillo.

PROBLEMA: Cadena dentro de cadena

ENUNCIADO: Se proporcionará inicialmente un número con la cantidad de juegos de prueba. Cada juego de prueba consistirá en dos palabras, sin espacios. El problema deberá decir cuantas letras deberán eliminarse de la segunda palabra para que forme la primera, sin alterar el orden de ninguna letra. En el caso de que no sea posible, mostrara el mensaje "IMPOSIBLE". El programa distinguirá entre mayúsculas y minúsculas.

Ejemplo: perro esta presente en emperro, pero PERRO no lo esta. El orden importa, perro no esta presente en errop.

EJEMPLO DE ENTRADA: Un número n, con la cantidad de juegos de prueba, seguido de dos líneas por cada juego de prueba. Ninguna palabra tendrá más de 100 letras.

EJEMPLO DE SALIDA: Numero de letras a eliminar o "IMPOSIBLE" si no es posible

Entrada	**Salida**
3 perro elperrodesanroque PERRO perroRamonRamirez CaJaMeLoT CaravanaJuezaMeLonTomate	12 IMPOSIBLE 16

Comentando el problema:

En este problema observamos, que **siempre la segunda cadena debe ser mayor o igual que la primera, sino es imposible ya por definición**.

Después, hay que fijarse que **si hay una posible solución, esta siempre será Tamaño cadena2 - Tamaño de cadena1**.

Para resolver este ejercicio, debemos **recorrer la cadena2**, desde la primera posición hasta el final, y usaremos un **contador para la cadena1**.

Primero buscaremos la primera letra de la cadena1 a través de la cadena2. Si la encontramos, a partir de la siguiente posición (modificando el contador de la cadena1) y buscaremos la segunda letra de la cadena1 a partir de la posición de cadena2 en la que estábamos y así con todas las letras. Si están todas las letras, tiene solución.

Si en algún momento, quedan más letras que buscar, que posiciones de la cadena2 por visitar, significara que no hay solución.

Código fuente:

```
#include <iostream>
#include <cstring>
using namespace std;

int cadenaDentroCadena(char *p1,char *p2){
    int tam1=strlen(p1); // Tamanyo de las palabras
    int tam2=strlen(p2);
    int contador=0;
    // Recorremos la segunda entera
    for(int i=0;i<tam2 && contador<tam1;i++){
        // Cada coincidencia, avanzamos una posición la primera
        if(p1[contador]==p2[i]){
            contador++;
        }
    }
    // Si ha llegado al final la primera, es que se podía formar
    if(contador==tam1){
        return tam2-tam1;
    }
    // Caso contrario, no se podía formar
    else{
        return -1;
    }
}
int main(){
    char pal1[101],pal2[102]; // Palabras a comparar, 100 caracteres
    int N;
    int res;
    char tmp;
    cin >> N; // Leemos el total de casos de prueba
    cin.get(tmp); // Caracter temporal
    for (int i=0;i<N;i++){
        // Leemos las palabras
        cin.getline(pal1,100,'\n');
        cin.getline(pal2,100,'\n');
        // Vemos cuantas hay que eliminar o si no se puede
        res=cadenaDentroCadena(pal1,pal2);
        if(res==-1){
            cout << "IMPOSIBLE"<< endl;
        }
        else{
            cout << res << endl;
        }
    }
    return 0;
}
```

PROBLEMA: Sopa de letras

ENUNCIADO : Se proporciona los valores n y m, seguidos de una sopa de letras de tamaño n por m. A continuación se nos presenta un número x, que va seguido de X palabras. Suponemos que esas palabras estarán en la tabla. Debemos de indicar en que posición se encuentra su origen. Si una palabra se encuentra mas de una vez, cogeremos la que este mas arriba (principio de la tabla) y si aun persiste el empate, elegiremos la mas a la izquierda. Las palabras podrán estar en vertical, horizontal y en las diagonales. La búsqueda de palabras no distinguirá mayúsculas de minúsculas.

EJEMPLO DE ENTRADA: Dos números n y m, seguidos de una tabla de n por m. Tras ello, un numero x seguido de x palabras. Tanto n como m serán menores de 40 y no habrá más de 15 palabras a comprobar.

EJEMPLO DE SALIDA: Listado de posición de inicio de cada palabra. La primera posición será la posición 1.

Entrada	**Salida**
6 5 aLoLo bLarh xFaxo aXBLM vasdf iFFGH 3 LOLO lala XAVi	1 2 4 4 3 1

Comentando el problema:

Este problema básicamente es simular una búsqueda de palabras en una sopa de letras, **probando todas las combinaciones**.

Primero, deberemos resolver el problema de la no distinción entre mayúsculas y minúsculas. La solución mas fácil es **simplemente pasar todo a mayúsculas** (o todo a minúsculas) y operar con ello.

Una vez solventado este problema definimos como resolverlo.

El problema nos dice que en caso de varias ocurrencias, debemos mostrar la que este mas arriba, y si persiste la duda, coger la que este mas a la izquierda. Esto puede solucionarse **recorriendo en ese orden** : Empezamos por la que esta mas arriba y mas a la izquierda (posición 1,1), luego probamos 1,2, 1,3… hasta llegar al final de línea, entonces probamos la posición 2,1, 2,2, 2,3… así hasta acabar la sopa de letras.

Con esta búsqueda, supondremos que cada posición es el origen de la palabra, y luego comprobaremos en todas direcciones (Horizontal, vertical, diagonal) si la palabra esta o no. Cuando este, ya hemos finalizado la búsqueda.

El programa nos dice que toda palabra pedida estará al menos una vez, así que **no hay que hacer ningún control especial por si la palabra no estuviera**.

Código fuente:

```
#include <cstring>
#include <iostream>
using namespace std;
// Variables globales para facilitar el código
char sopa[55][55]; // Tabla para la sopa
char pal[55]; // Palabra que buscaremos
// Tamaño de la sopa
int m,n;

/* Función que recibe una una posición
y devuelve si en esa posición comienza o no esa palabra*/
int esta_pal(int y,int x){
    int i,j;
    // Tamaño de la palabra
    int tampal=strlen(pal);
    // Resultado, por defecto 0 (no encontrado)
    int res=0;
    // Si el primer carácter no coincide, no buscamos
    if(pal[0]!=sopa[y][x]){
        return 0;
    }
    /** Izquierda **/
    if(x-tampal+1>=0 && res==0){
       res=1;// Por defecto, consideramos que esta
       for(i=x-1;i>=x-tampal+1 && res==1;i--){
            if(pal[x-i]!=sopa[y][i])
                res=0;// Si descubrimos no esta, paramos de buscar
       }
    }

    /** Derecha **/
    if(x+tampal-1<n && res==0){
       res=1;// Por defecto, consideramos que esta
       for(i=x+1;i<x+tampal && res==1;i++){
            if(pal[i-x]!=sopa[y][i])
                res=0;// Si descubrimos no esta, paramos de buscar
       }
    }

    /** Arriba **/
    if(y-tampal+1>=0 && res==0){
       res=1;// Por defecto, consideramos que esta
       for(i=y-1;i>=y-tampal+1 && res==1;i--){
            if(pal[y-i]!=sopa[i][x])
```

```
                res=0;// Si descubrimos no esta, paramos de buscar
            }
        }

        /** Abajo **/
        if(y+tampal-1<m && res==0){
            res=1;// Por defecto, consideramos que esta
            for(i=y+1;i<y+tampal && res==1;i++){
                if(pal[i-y]!=sopa[i][x])
                    res=0;// Si descubrimos no esta, paramos de buscar
            }
        }

        /** Abajo izquierda **/
        if(y+tampal-1<m  && x-tampal+1>=0 && res==0){
            res=1;// Por defecto, consideramos que esta
            for(i=y+1,j=x-1;i<y+tampal  &&  j>=x-tampal+1  &&  res==1;i++,j--
){
                if(pal[i-y]!=sopa[i][j])
                    res=0;// Si descubrimos no esta, paramos de buscar
            }
        }

        /** Abajo derecha **/
        if(y+tampal-1<m && x+tampal-1<n  && res==0){
            res=1;// Por defecto, consideramos que esta
            for(i=y+1,j=x+1;i<y+tampal  && j<x+tampal && res==1;i++,j++){
                if(pal[i-y]!=sopa[i][j])
                    res=0;// Si descubrimos no esta, paramos de buscar
            }
        }

        /** Arriba izquierda **/
        if(y-tampal+1>=0 && x-tampal+1>=0  && res==0){
            res=1;// Por defecto, consideramos que esta
            for(i=y-1,j=x-1;i>=y-tampal+1  &&  j>=x-tampal+1  &&  res==1;i--
,j--){
                if(pal[y-i]!=sopa[i][j])
                    res=0;// Si descubrimos no esta, paramos de buscar
            }
        }

        /** Arriba derecha **/
        if(y-tampal+1>=0 && x+tampal-1<n  && res==0){
            res=1; // Por defecto, consideramos que esta
            for(i=y-1,j=x+1;i>=y-tampal+1  &&  j<x+tampal  &&  res==1;i--
,j++){
                if(pal[y-i]!=sopa[i][j])
                    res=0; // Si descubrimos no esta, paramos de buscar
            }
```

```
    }
    return res;
}

int main() {
    int i,j,k;
    // Variables para las posiciones x,y de las palabras
    int xpal,ypal;
    int npab; // Cuantas palabras habrá
    cin >> m >> n; // Leemos m y n
    char tmp;
    // Leemos carácter temporal, para luego poder leer líneas
completas
    cin.get(tmp);
    // Leemos las filas, y las pasamos a mayúsculas.
    for(int i=0;i<m;i++){
        cin.getline(sopa[i],51,'\n');
        strupr(sopa[i]);
    }
    // Leemos cuantas palabras
    cin >> npab;
    // Carácter temporal, para leer luego líneas completas
    cin.get(tmp);
    // Leemos todas las palabras
    for(i=0;i<npab;i++){
        xpal=0, ypal=0;
        // Centinela, indicando que en principio no se ha encontrado
        int encontrado=0;
        // Leemos palabra y la pasamos a mayúsculas
        cin.getline(pal,51,'\n');
        strupr(pal);
     // Buscamos en el orden  para encontrar la mas arriba/izquierda
        // si en esa posición empieza la palabra que tenemos
        // El centinela se pone a uno y acaba la búsqueda
        for(j=0;j<m && !encontrado;j++){
            for(k=0;k<n && !encontrado;k++){
                // Función que devuelve si esta o no la palabra
                if(esta_pal(j,k)){
                    xpal=j;
                    ypal=k;
                    encontrado=1;
                }
            }
        }
        // Imprimimos la posición de la palabra
        cout << xpal+1 << " " << ypal+1 << endl;
    }
    return 0;
}
```

CAPÍTULO VI: Problemas de ordenación

Existen multitud de problemas, cuya solución consiste en ordenar algo, o parte de su solución requiere un algoritmo de ordenación.

Los algoritmos de ordenación son un tema de estudio e investigación. Posiblemente los algoritmos de ordenación hayan sido los algoritmos más importantes descubiertos y aplicados en la informática moderna. Existen muchos tipos, con ventajas e inconvenientes.

No es el fin de este libro implementarlos uno a uno y compararlos (podría hacerse varios libros solo sobre ese tema), sino guiar a conocer su existencia y su uso en la practica.

Algoritmos de ordenación más conocidos:

- **InsertionSort**
- **SelectionSort**
- **ShellSort**
- **MergeSort**
- **QuickSort**

Podéis investigar en multitud de sitios por Internet sobre estos algoritmos y aprender más sobre ellos.

A grandes rasgos y para la mayoría de contextos el mas rápido es **QuickSort** aunque según el caso o según el contexto esta elección puede variar (memoria, tiempo, características de los datos).

Una de las grandes ventajas, es que muchos de los algoritmos de ordenación ya están implementados en las bibliotecas de los lenguajes de programación mas utilizados.. En los siguientes ejemplos, usaremos la implementación de QuickSort en C, que nos ayudara a aislarnos de codificar el algoritmo y centrarnos en saber que queremos ordenar y como.

PROBLEMA: Anagrama

ENUNCIADO : Un anagrama consiste en dos palabras distintas que se forman exactamente usando las mismas letras. Por ejemplo AMOR y ROMA son dos anagramas. Se proporcionara inicialmente un número con la cantidad de juegos de prueba. Cada juego de prueba consistirá en dos palabras, sin espacios.

El problema deberá decir cuantas letras deberán eliminarse de la segunda palabra para que sea un anagrama de la anterior palabra. Para facilitar el programa, se aceptara como anagrama la misma palabra. En el caso de que no sea posible, mostrara el mensaje "IMPOSIBLE". El programa distinguirá entre mayúsculas y minúsculas.

Ejemplo: amor y promar, hay que eliminar una letra para formar el anagrama.

EJEMPLO DE ENTRADA: Un número n, con la cantidad de juegos de prueba, seguido de dos líneas por cada juego de prueba. Ninguna palabra tendrá más de 100 letras.

EJEMPLO DE SALIDA: Numero de letras a eliminar o "IMPOSIBLE" si no es posible.

Entrada	**Salida**
4 amor proma amor Roma CaJa MariJaCa Raymagini imaginaRy	1 IMPOSIBLE 4 0

Comentando el problema:

Este problema es muy parecido al que vimos en el capítulo anterior "Cadenas dentro de Cadenas". Realmente, el mismo problema, la única diferencia es que debemos ordenar antes las cadenas. Si recordáis, en el primer problema importaba el orden, pero aquí no (por ser anagramas).

Por eso en el anterior no ordenábamos (manteníamos relación de orden inicial) y en este si ordenamos, estableciendo una nueva relación de orden (orden alfabético).

Para hacer esto rápido, hemos utilizado la implementación de **QuickSort** en C, mediante la función **qsort** que esta dentro de la biblioteca **stdlib**.

Su sintaxis es la siguiente :

```
qsort(datos, tamanyo, sizeof(tipodedato),
&comparacion);
```

Donde **datos** es el vector a ordenar, **tamanyo** es el tamaño del vector, **sizeof(tipodedato)** es el tamaño en bytes del tipo de dato (es fácil conseguirlo con sizeof y el dato, por ejemplo sizeof(int)) y por ultimo **comparación** es el nombre de una función que contendrá el código para decidir si un dato va delante o detrás de otro.

Si esta función de comparación **devuelve negativo**, es que el primer elemento va antes que el segundo. **Si devuelve positivo**, el segundo elemento va antes que el primero. **Si devuelve 0**, ambos elementos son iguales y tienen el mismo orden.

Un ejemplo de esa función para ordenar enteros:

```
int comparacion(const void *_a, const void
*_b) {

        int *a, *b;

        a = (int *) _a;
        b = (int *) _b;

        return (*a - *b);
}
```

Aquí tenéis una solución del problema usando **qsort** para ordenar cadenas en orden lexicográfico:

Código fuente:

```
#include <iostream>
#include <cstring>
#include <cstdlib>

using namespace std;

// Función usada en Qsort, compara dos letras y devuelve su relación
de orden
int comparar
(const void *_a, const void *_b) {
        char *a, *b;
        a = (char *) _a;
        b = (char *) _b;
        return (*a - *b); // Basándose valor ASCII de las letras
}

int anagrama(char *p1,char *p2){
    int tam1=strlen(p1); // Tamanyo de las palabras
    int tam2=strlen(p2);

    // Las ordenamos las palabras

    qsort(p1, tam1, sizeof(char), &comparar);
    qsort(p2, tam2, sizeof(char), &comparar);

    // Al ordenar, el problema se reduce a ver si se puede formar o
no

    int contador=0;
    // Recorremos la segunda entera
    for(int i=0;i<tam2 && contador<tam1;i++){
        // Cada coincidencia, avanzamos una posición la primera
        if(p1[contador]==p2[i]){
            contador++;
        }
    }
    // Si ha llegado al final la primera, es que se podía formar
    if(contador==tam1){
        return tam2-tam1;
    }
    // Caso contrario, no se podía formar
    else{
        return -1;
    }
```

```
}


int main(){
    char pal1[101],pal2[102]; // Palabras a comparar, 100 caracteres
    int N;
    int res;
    char tmp;
    cin >> N; // Leemos el total de casos de prueba

    cin.get(tmp); // Caracter temporal
    for (int i=0;i<N;i++){
        // Leemos las palabras
        cin.getline(pal1,100,'\n');
        cin.getline(pal2,100,'\n');
        // Vemos cuantas hay que eliminar o si no se puede
        res=anagrama(pal1,pal2);
        if(res==-1){
            cout << "IMPOSIBLE"<< endl;
        }
        else{
            cout << res << endl;
        }
    }
    return 0;
}
```

PROBLEMA: Ordenación de enteros

ENUNCIADO: Realiza un programa que leerá hasta un máximo de 100.000 enteros (un entero por línea) y deberá imprimir dicha lista de enteros en orden.

EJEMPLO DE ENTRADA: hasta 100.000 enteros, cada uno por línea.

EJEMPLO DE SALIDA: Lista de enteros ordenados, uno por línea en orden.

Entrada	**Salida**
5 3 6 4 7 11 22 11111 -1 22	-1 3 4 5 6 7 11 22 22 11111

Comentando el problema:

Para solucionar este problema, lo primero a tener en cuenta es que si te dicen que la entrada será como máximo de **100.000 enteros, significa que algún caso de prueba tendrá 100.000 enteros**.

Otra cosa es fijarse bien en que tipos de datos piden, aquí son enteros, eso implica números negativos también (aunque no afecta a la solución del problema).

También hay que tener claro como se construye la relación de orden (esa dificultad la veremos en el siguiente problema propuesto).

Este problema podemos resolverlo fácilmente, leyendo la entrada, usando la función de ordenación **qsort** de C y finalmente imprimiendo el contenido del vector ordenado. Es otro ejemplo práctico del uso de esta función.

Código fuente:

```
#include <iostream>

#include <cstring>
#include <cstdlib>

using namespace std;

// Vector para almacenar 100000 enteros
int enteros[100000];

// Función usada en Qsort, compara dos enteros y devuelve su
relación de orden
int comparar
(const void *_a, const void *_b) {
        int *a, *b;
        a = (int *) _a;
        b = (int *) _b;
        return (*a - *b);
}

int main(){
    int tam=0;
    while(cin >> enteros[tam]){
        tam++;   //   Incrementamos   tam   para   guardar   el
siguiente
    }
    // Ordenamos
    qsort(enteros, tam, sizeof(int), &comparar);
    // Imprimimos el vector ordenado
    for(int i=0;i<tam;i++){
        cout << enteros[i]<<endl;
    }
    return 0;
}
```

PROBLEMA: Ordenación de enteros por valor absoluto

ENUNCIADO: Realiza un programa que leerá hasta un máximo de 100.000 enteros (un entero por línea) y deberá imprimir dicha lista de enteros en orden. Los enteros se ordenarán según su valor absoluto. Valor absoluto es el valor de un entero, prescindiendo de su signo.

EJEMPLO DE ENTRADA: hasta 100.000 enteros, cada uno por línea.

EJEMPLO DE SALIDA: Lista de enteros ordenados por valor absoluto, uno por línea en orden.

Entrada	**Salida**
5 -3 6 4 -7 11 22 11111 -1 22	-1 -3 4 5 6 -7 11 22 22 11111

Comentando el problema:

Aquí la cosa ya cambia, ya que **el valor no es el habitual de los enteros, sino que el orden se realizará por valor absoluto**.

A muchas personas se les ocurren ideas para resolver este problema, que sin ser descabelladas, no funcionan. Vamos a repasar algunas ideas para ver sus inconvenientes.

1) Una posible aproximación es pasar los datos a valor absoluto. Esa aproximación no vale, ya que hay que mostrar los datos y perderíamos esa información.

2) Otra posible ordenación es usar un vector con datos en valor absoluto y otro sin, y ordenar respecto al que tiene los datos en valor absoluto. Esta solución es posible, e incluso podría ser aceptada en este problema concretamente, pero no es la mejor solución ya que consumiría el doble de memoria.

La mejor solución para afrontar el problema es manipular la función de comparación. Esa función solo dice si un elemento va antes que otro devolviendo negativo, positivo o cero. Pero por dentro puede ser como nosotros queramos. Haremos que esa función pase los datos a valor absoluto y entonces compare. Eso no alterara los datos originales.

Aquí tenéis una solución al problema:

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

// Vector para almacenar 100000 enteros
int enteros[100000];
// Función usada en Qsort, compara dos enteros y devuelve su relación
de orden
int comparar
(const void *_a, const void *_b) {
        int *a, *b;
        int tmpa,tmpb;
        a = (int *) _a;
        b = (int *) _b;

        if(*a<0){ // Pasamos a valor absoluto al comparar
            tmpa=-1*(*a);
        }
        else{
            tmpa=*a;
        }
        if(*b<0){// Pasamos a valor absoluto al comparar
            tmpb=-1*(*b);
        }
        else{
            tmpb=*b;
        }
        // Comparamos los valores absolutos
        return (tmpa - tmpb);
}
int main(){
    int tam=0;
    while(cin >> enteros[tam]){
        tam++; // Incrementamos tam para guardar el siguiente
    }
    // Ordenamos por valor absoluto
    qsort(enteros, tam, sizeof(int), &comparar);
    // Imprimimos el vector ordenado
    for(int i=0;i<tam;i++){
        cout << enteros[i]<<endl;
    }
    return 0;
}
```

PROBLEMA: Ordenación por peso y altura

ENUNCIADO: Realiza un programa que leerá hasta un máximo de 100.000 personas con su peso y altura. Cada persona se definirá por un nombre (único y sin espacios) seguido de dos enteros, correspondientes al peso en kilogramos y a la altura en centímetros. El programa deberá mostrar una lista de nombres (uno por línea) ordenados de mayor a menor por peso y en caso de empate, ordenados por altura.

EJEMPLO DE ENTRADA: hasta 100.000 personas, formada por nombre, peso y altura. El nombre no superará los 100 caracteres.

EJEMPLO DE SALIDA: Lista de personas (una por línea) ordenadas por peso de mayor a menor y en caso de empate ordenadas por altura de mayor a menor.

Entrada	**Salida**
Pedro 100 191 Jose 100 199 Lato 80 166 Miki 77 188 Marcos 88 199 Mercedes 77 180	Jose Pedro Marcos Lato Miki Mercedes

Comentando el problema:

En este problema ya usamos dos datos para ordenar, no un solo dato como los anteriores. Esto complica un poco las cosas, pero usando herramientas del lenguaje C/C++ podemos solventar este inconveniente y hacer una solución del mismo estilo que las anteriores.

Una posible solución es usar struct combinado con typedef de C/C++ para crear un estructura o tipo de datos nuevo y entonces usar QSort como siempre (usando el tipo de datos creado).

Dentro de la función de ordenación, tendremos en cuenta primero el peso, y en caso de empate tendremos en cuenta la altura.

Un ejemplo del uso de typedef y struct en C/C++:

```
/* Define un tipo punto, que es una
Estructura formada por dos enteros */
typedef struct {
   int    x;
   int    y;
} punto;

/* Inicializa una variable de tipo punto */
punto p = {1,2};
p.x=4;
```

Código fuente:

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
typedef struct {
    char nombre[101];
    int peso;
    int altura;
} pesoAltura;
// Vector para almacenar 100000 pares peso/altura
pesoAltura valores[100000];
// Función usada en Qsort, compara enteros y devuelve su orden
int comparar(const void *_a, const void *_b) {
        pesoAltura *a, *b;
        pesoAltura tmpa, tmpb;
        a = (pesoAltura *) _a;
        b = (pesoAltura *) _b;
        tmpa=*a;
        tmpb=*b;
        /* Ordenamos de mayor a menor, fijaros orden de la resta*/
        // Pesos diferentes, decide el peso, sino la altura
        if(tmpa.peso!=tmpb.peso){
            return tmpb.peso-tmpa.peso;
        }
        else{
            return tmpb.altura-tmpa.altura;
        }
}
int main(){
    int tam=0;
    char tmp; // Para leer carácter temporal salto de línea
    while(cin >> valores[tam].nombre){
        cin >> valores[tam].peso;
        cin >> valores[tam].altura;
        // Leemos carácter temporal salto de línea
        cin.get(tmp);
        tam++; // Incrementamos tam para guardar el siguiente
    }
    // Ordenamos por valor absoluto
    qsort(valores, tam, sizeof(pesoAltura), &comparar);
    // Imprimimos el vector ordenado
    for(int i=0;i<tam;i++){
        cout << valores[i].nombre <<endl;
    }
    return 0;
}
```

CAPÍTULO VII: Problemas de fuerza bruta

Existen multitud de problemas en el que debemos encontrar una o varias soluciones que cumpla unas determinadas características. Estos son problemas resolubles mediante **la fuerza bruta** (o lo que es lo mismo, probar combinaciones).

Muchas veces, la única forma de resolver estos problemas, **es generar todas las posibles soluciones y comprobar si son validas o no**. Esta técnica en algunos casos es útil, aunque en otros el número de soluciones a probar es muy grande. En esos casos hay que buscar optimizaciones para reducir el número de búsquedas sin que la solución sea correcta.

Veremos algunos ejemplos de problemas de fuerza bruta y propondremos algunas soluciones.

El concepto de forma abstracta es difícil de entender, pero con los ejemplos que veremos a continuación y los problemas propuestos, espero quede claro el concepto.

Imaginemos nos dan 2 cifras y nos dicen que debemos usar todas las cifras en cualquier orden, y que con ellas podemos hacer la operación que queramos entre sumar y restar y nos piden saber de cuantas formas distintas se puede sacar un resultado concreto.

Ejemplo:

Cifras: 3 y 6

Resultado a buscar: 3

Para saber cuantas combinaciones dan 3, deberemos probar todas las posibles combinaciones, obtener el resultado de esa combinación y ver si efectivamente el resultado es 3.

Combinaciones:

3 + 6 = 9 // No es tres

- 3 + 6 = 3 // **Si es tres**

3 - 6 = 9 // No es tres

- 3 - 6 = -9 // No es tres

6 + 3 = 9 // No es tres

- 6 + 3 = -3 // No es tres

6 - 3 = 3 // **Si es tres**

- 6 - 3 = -9 // No es tres

Observando todas las combinaciones (8 en total), vemos que hay 2 que dan como resultado 3.

Como el enunciado decía que se podía usar cualquier orden, se han tenido en cuenta las operaciones "Espejo". Estas podrían haberse ahorrado aprovechado la propiedad conmutativa de la suma.

En ese caso, cada vez que se encontrara un resultado, se sumarian dos y solo se probarían una vez, sin espejo.

Combinaciones:

3 + 6 = 9 // No es tres

- 3 + 6 = 3 // Si es tres (Y sumamos dos)

3 - 6 = 9 // No es tres

- 3 - 6 = -9 // No es tres

Así hemos **reducido el número de combinaciones a probar a la mitad.**

PROBLEMA: Suma de dos números

ENUNCIADO: Realiza un programa que leerá dos números. Estos podrán sumarse o restarse en cualquier orden. Tras ello leerá un número que será el resultado deseado. El programa deber indicar cuantas combinaciones dan como resultad el numero deseado.

EJEMPLO DE ENTRADA: dos números (uno por línea) con los números a combinar, mas otro número en otra línea, con el resultado deseado.

EJEMPLO DE SALIDA: numero de combinaciones que dan el resultado deseado.

Entrada	**Salida**
3 6 3	2

Comentando el problema:

Usamos la explicación anterior.

Código fuente:

```
#include <iostream>


using namespace std;

int main(){
    int x,y,res;
    int contador=0;
    cin >> x >> y >> res; // Leemos los datos
    /* Comprobamos las combinaciones para ver si coinciden con el
resultado  Además, tenemos en cuenta el efecto espejo para reducir
combinaciones*/
    if(x+y==res){
        contador+=2;
    }
    if(-x+y==res){
        contador+=2;
    }
    if(x-y==res){
        contador+=2;
    }
    if(-x-y==res){
        contador+=2;
    }
    cout << contador << endl; //Mostramos el resultado
    return 0;
}
```

La solución **es bastante simple**, son cuatro if. Pero esto es posible porque **el problema no era muy grande** (solo 4 combinaciones). En algunos casos, el número de combinaciones a probar será elevado. Y en otros, el problema tendrá una **entrada variable**. (Imaginaos el caso que en vez de ser siempre dos números, fueran n (n entre 2 y 5) números).

PROBLEMA: Suma de N números

ENUNCIADO: Realiza un programa que leerá un numero N y después tantos números como el valor de N sea. Estos podrán sumarse o restarse en cualquier orden. Tras ello leerá un número que será el resultado deseado. El programa deber indicar cuantas combinaciones dan como resultad el numero deseado.

EJEMPLO DE ENTRADA: un número N y N números (uno por línea) con los números a combinar, mas otro número en otra línea, con el resultado deseado. N será un número entre 2 y 5 ambos inclusive)

EJEMPLO DE SALIDA: numero de combinaciones que dan el resultado deseado.

Entrada	**Salida**
2 3 6 3	2

Comentando el problema:

Aquí la cosa se complica un poco. **Aquí si que generaremos todas las combinaciones posibles y veremos si encajan o no, sin usar ninguna optimización como la del espejo**. Al ser 5 el número máximo a combinar y este no ser excesivamente grande, podremos usar la **recursividad** para hacer el código más sencillo.

El termino recursividad, se refiere a cuando una función desde su código se llama una o varias veces a si misma.

En la solución**, a la función recursiva** le pasaremos cuantos números sumados llevamos y cual es el resultado parcial. Además, para no repetir el uso de números, le pasaremos un vector con valores boléanos (0 y1) indicando si un numero ha sido usado ya en la combinación o no. Cuando se usen todos los números, se comprobara si el valor suma lo deseado. En ese caso, se aumentara en 1 un contador que finalmente será impreso por pantalla.

Código fuente:

```
#include <iostream>

using namespace std;

int numeros[5];
int usados[5];
int N,res;
int contador;

/* Función que recibe valor a sumar, "cantidad de números usados",
, el total de suma que llevamos por ahora y un vector
indicando que números se han usado
*/
void calcular(int valor, int cant,int total,int usa[]){
    // Acumulamos el valor al total
    int t=total+valor;
    int c=cant+1; // Sumamos un numero mas usado
    if(c==N){
        if(t==res){
            contador++;
        }
        return;
    }

    for (int i=0;i<N;i++)
    {
        // Si un numero no se ha usado...
        if(!usa[i]){ // Distinto de cero
           usa[i]=1; //Marcamos como usado
           calcular(numeros[i],c,t,usa);
           calcular(-numeros[i],c,t, usa);
           usa[i]=0; // Desmarcamos como usado
        }
    }

}

int main(){

    contador=0;
    cin >> N; // Leemos cantidad de numeros
    // Leemos cada numero y marcamos como no usado
    for (int i=0;i<N;i++){
        cin >> numeros[i];
        usados[i]=0;
    }
    cin >> res;
```

```
    for (int i=0;i<N;i++)
    {
        usados[i]=1; // Marcamos numero como usado

        // llamamos a la función con valores iniciales
        // y tanto en positivo como en negativo

        // Caso base, 0 usados, 0 total

        calcular(numeros[i],0,0,usados);
        calcular(-numeros[i],0,0, usados);
        // Desmarcamos numero como usado
        usados[i]=0;
    }
    cout << contador << endl; //Mostramos el resultado
    return 0;
}
```

PROBLEMA: Cuantos DNIs

ENUNCIADO: Realiza un programa que leerá un DNI. Ese DNI constara de 8 dígitos y una letra, aunque algunos dígitos podrán estar sustituidos por un valor X. Suponiendo que X pueda sustituirse por cualquier digito, indicar cuantos DNIs distintos pueden formarse con el número de DNI proporcionado. Recordad algoritmo de la letra del DNI

Algoritmo de cálculo del DNI

Usamos nuestro numero de DNI y le aplicamos la operación "Modulo 23" (El número obtenido, se busca su correspondencia en esta tabla, y de ahí obtenemos la letra.

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

No se utilizan las letras: I, Ñ, O, U

EJEMPLO DE ENTRADA: un número N y N DNIs (uno por línea), formado por 8 dígitos mas la letra (los dígitos podrán ser sustituidos por una X)

EJEMPLO DE SALIDA: numero de combinaciones que dan un DNI valido.

Entrada	**Salida**
4 12345678G 12345678Z X23456XXM 1X3X5X7XR	0 1 43 1686

Comentando el problema:

Para solucionar este problema, lo que hacemos es hacer una función recursiva que cuando encuentra una X, hace diez llamadas a si misma, sustituyendo la X por los dígitos del 0 al 9.

Esto lo hace por cada X que encuentra, **generando así todas las combinaciones posibles**.

Cuando llega al final de una combinación, comprueba si el DNI es válido o no. En caso de ser válido, suma uno a la solución.

La función recursiva va acumulando las sumas parciales (soluciones parciales) de sus llamadas recursivas, devolviendo en su llamada original la solución.

La función recursiva "calcular" se puede observar y estudiar detenidamente, observando que se pasa un DNI, la posición desde donde empieza a buscar y cual es la última posición. La función devuelve la suma de soluciones acumulada.

Código fuente:

```
#include <iostream>
#include <cstring>
using namespace std;

// Posición 9, es la letra
int dniValido(char d[9]){
    // Correspondencias letra/numero
    char letra=d[8]; // Sacamos la letra
    char ntemp[9];
    // Pasamos a un temporal el numero sin letra
    // Para usar ATOI
    for (int i=0;i<8;i++){
        ntemp[i]=d[i];
    }
    ntemp[8]='\0';
    // Cadena a entero
    int ndni=atoi(ntemp);
    // Algoritmo calculo de DNI
    char tabla[] = "TRWAGMYFPDXBNJZSQVHLCKE";
    char letrareal=tabla[ndni%23];
    // Si la letra real coincide con la que tenemos
    if(letra==letrareal){
        return 1; // DNI correcto
    }
    return 0; // Dni incorrecto

}

int calcular(char d[9],int posact,int tam){
    int i,j;
    int res=0;
    if(posact==tam){
        // Miramos si el dni generado es valido
        if(dniValido(d)){
            return 1;
        }
        return 0;
    }
    // Contamos si ha habido X o no
    int cuantasX=0;
    // Empezamos desde la posición enviad
    // La letra no hay que tenerla en cuenta, por eso tam-1
    for(i=posact;i<tam-1;i++){
        if(d[i]=='X'){ // si la posición es sustituible
            cuantasX=1; // Marcamos como que ha habido X
            for(j=0;j<=9;j++){
                d[i]='0'+j; // Truco para pasar int a char
```

```
                // Función recursiva
                res=res+calcular(d,i,tam); // Calculamos a partir de
la posición siguiente
                d[i]='X'; // Restauramos la X, para seguir generando
combinaciones
            }
        }
    }
    // Caso de ninguna X, para ver si el dni que tenemos es valido
    if(cuantasX==0){
        // Miramos si el dni generado es valido
        if(dniValido(d)){
            return 1; // Devolvemos 1 si el dni es valido y no había
X
        }
    }
    // Devolvemos la cantidad encontrada
    return res;
}


int main(){
    char DNI[10];
    int resultado,N;
    cin >> N; // Leemos cantidad de números
    // Leemos cada numero y marcamos como no usado
    for (int i=0;i<N;i++){
        cin >> DNI;
        // Calculamos con la función recursiva e imprimimos
        resultado=calcular(DNI,0,strlen(DNI));
        cout << resultado << endl;
    }
    return 0;
}
```

CAPÍTULO VIII: Colas, Colas de prioridad, pilas y diccionarios

Debemos hablar de algunas estructuras muy útiles en programación**: las colas, las pilas y los diccionarios**.

La cola es una estructura de tipo FIFO (First In First Out, o en castellano Primero que entra, primero que sale). El funcionamiento es similar a cualquier cola del cine, de supermercado. Las peticiones, llegan en un orden y se atienden en orden de llegada. El que llega primero se atiende el primero, y el último será atendido en último lugar.

La pila es una estructura tipo LIFO (Last In, First Out o en castellano Ultimo que entra, primero que sale). Imaginemos un montón de libros apilados (uno encima del otro). El funcionamiento de la pila, puede es similar a apilar cosas (libros, paquetes, etc.). En el caso de libros apilados, el primero que sacaremos, será el último que hayamos puesto (el que esta arriba del todo de la pila).

El diccionario : es una estructura que funciona como un diccionario, guarda una clave asociada a un valor. Ejemplo la clave “pesoJuan” tiene

asociado un 88 y la clave “pesoPedro” tiene asociado un 99. Es una forma rápida de introducir/ consultar información. En un diccionario corriente seria que a una palabra le asigna una definición.

Dentro de los diccionarios, hay una versión más simple de estos, llamada **conjunto**. En los conjuntos, solo se almacenan claves, sin definición (sirven para saber si dicha clave esta o no dentro del conjunto).

Estas estructuras pueden ser implementadas manualmente de distintos modos (podéis recabar mas información en Internet).
En C++ ya están implementadas mediante la STL, que además es nativa de C/C++ (esta disponible en los jueces) y podéis usarlas fácilmente para vuestros ejercicios. Las presentadas aquí son a modo de introducción, la STL implementa muchas mas estructuras que pueden ser útiles. Para más información **http://www.sgi.com/tech/stl/index.html .**

A continuación un par de ejemplos de implementación de colas, pilas y diccionario usando **STL**.

Ejemplo Cola:

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
    // Declaramos dos colas con tipo interior string
    queue<string> colamayu, colaminu;
    // Usamos el tipo string de C++
    string cad;
    while (cin >> cad) {
        // Leemos palabras y las clasificamos en dos colas según
empiecen
        // por mayúsculas o minúsculas
        if (cad[0]>='A' and cad[0]<='Z'){
            colamayu.push(cad);
        }
        else{
            colaminu.push(cad);
        }
    }
    cout << "Hay " << colamayu.size()<< " palabras que empiezan por
mayusculas"<<endl;
    while (!colamayu.empty()) {// Mientras haya algo en la cola
        cout << colamayu.front() << endl; // Imprimimos la actual
        colamayu.pop(); // Sacamos de la cola
    }
    cout << "Hay " << colaminu.size()<< " palabras que empiezan por
minusculas"<<endl;

    while (!colaminu.empty()) {// Mientras haya algo en la cola
        cout << colaminu.front() << endl; // Imprimimos la actual
        colaminu.pop(); // Sacamos de la cola
    }
}
```

En este ejemplo, vemos como se usan dos colas para guardar palabras que empiezan por mayúscula o minúscula, y luego se desencolan.

Ejemplo Pila:

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
    // Declaro una pila de enteros
    stack<int> pila;
    int num;
    while (cin >> num){
        // Apilo el entero leído
        pila.push(num);
    }
    // Mientras la pila no este vacía
    while (!pila.empty()) {
        // Muestro el que esta arriba de la pila
        cout << pila.top()<<endl;
        // Desapilo el que esta arriba
        pila.pop();
    }
}
```

En este ejemplo vemos como se apilan números enteros y luego se extraen de la pila y se muestran en orden inverso al introducido.

Ejemplo Diccionario:

```
#include <iostream>
#include <map>

using namespace std;


int main(){
    // Definimos un diccionario, cuya clave es un string y
    // su contenido es otro string
    map<string, string> jugadores;
    // Con la clave Futbol, metemos el contenido Messi
    jugadores["Futbol"] = "Messi";
    // Con la clave Basket, metemos el contenido Gasol
    jugadores["Basket"] = "Gasol";
    cout << "De basket: " << jugadores["Basket"] << endl;
    cout << "De futbol: " << jugadores["Futbol"] << endl;
}
```

En este ejemplo vemos como se introducen en un diccionario las claves “Futbol” y “Basket” y a cada una se le asocia un jugador famoso (“Messi” y “Gasol”).

PROBLEMA: Cierre de paréntesis

ENUNCIADO: El programa recibirá una cadena, con un conjunto de paréntesis (o) y deberá indicar si están bien cerrados o no.

) Mal cerrado

() Bien cerrado

(() Mal cerrado

)(Mal cerrado

EJEMPLO DE ENTRADA: Cadena de paréntesis, de cómo máximo 100 caracteres.

EJEMPLO DE SALIDA: SI en caso de estar bien cerrado, NO en caso de estar mal cerrado.

Entrada	**Salida**
(()()(()))	SI
(()))()	NO

Este problema puede ser resuelto muy fácilmente **mediante una pila**.

Cada vez que haya un "(", apilamos. Si hay un ")", des-apilamos. Si intentamos desafilar y no podemos (pila vacía), es que hay un desequilibrio y esta mal cerrado,

Al finalizar de recorrer los paréntesis, si aun queda alguno por des-apilar, es que le falta un cierre a derechas, por lo cual es incorrecto.

En caso que no quede ninguno, es que el paréntesis estaba cerrado correctamente.

Aquí una solución **usando la pila de la STL**.

Código fuente:

```
#include <iostream>
#include <stack>
#include <cstring>
using namespace std;

int main() {
    char parentesis[102];
    int tam;
    // Declaro una pila de caracteres
    stack<char> pila;
    cin >>  parentesis;
    tam=strlen(parentesis);
    // Recorremos todos los caracteres
    for(int i=0;i<tam;i++){
        // si es ( , apilamos
        if(parentesis[i]=='('){
            pila.push(parentesis[i]);
        }
        // Si es ) , desapilamos
        else{
            if(pila.size()>0){
                pila.pop();
            }
            // Si no podemos desapilar porque no quedan, mal cerrado
            else{
                cout << "NO" << endl;
                return 0;
            }
        }
    }
    // si al finalizar, queda alguno por cerrar, mal cerrado
    if(pila.size()>0){
        cout << "NO" << endl;
    }
    else{
        cout << "SI" << endl;
    }
    return 0;
}
```

CAPÍTULO IX: Ampliar conocimientos

El mundo de la programación, la algorítmica y los concursos de programación **es muy grande** y nos dejamos muchas cosas en el tintero. Hay gran cantidad de temas a explorar y desde aquí hago algunas sugerencias donde recomiendo profundizar.

- **Algoritmos de grafos**
    - Concepto de grafos
    - Búsqueda en anchura (Camino mas corto sin pesos)
    - Búsqueda en profundidad (Tamaño de un grafo, conectividad)
    - Árbol mínimo expandido
    - Algoritmo de Dijkstra (Camino mas corto con pesos)
    - Floyd Warshall (Todos los caminos mínimos)
    - Caminos Eulerianos (Pasando por todos los aristas)
- **Combinatoria**
    - Técnicas de recuento
    - Coeficientes binomiales (ver de K elementos, de cuantas formas se pueden elegir N distintos (N<=K))

- **Programación dinámica** (Evitar cálculos repetitivos y reducir tiempo de ejecución)
- **Geometría**
    - Áreas, perímetros
    - Convex Hull (Recubrimiento mínimo de todos los puntos)

Aquí dejo una serie de enlaces que espero os ayuden en la tarea de ampliar conocimiento.

Libros

- **Concursos Internacionales de Informática Y Programación**
- **Art of programming contest**
- Enlaces a **libros gratuitos** relacionados con los concursos de programación: http://www.icpc-bolivia.edu.bo/libros.html

Enlaces

- **Pagina de entrenamiento del equipo USA para la IOI:** http://www.usaco.org/
- **Pagina oficial de la olimpiada informática española**: http://olimpiada-informatica.org/

- **Codebreakers** (blog sobre algorítmica) :

  http://codebreakerscorp.wordpress.com/

- **Referencia biblioteca STL** :

  http://www.sgi.com/tech/stl/index.html

- **Methods to solve** (pagina sobre como resolver problemas de la ACM-UVA) :

  http://www.comp.nus.edu.sg/~stevenha/programming/acmoj.html

- **Hispabyte**: foro de discusión en castellano sobre concursos de programación http://hispabyte.net