
RHG DOR FQG

Laboratorio

MADRID 2017

Índice general

| | | |
|----------|--|-----------|
| I | Fundamentos | 1 |
| 1 | Introducción | 2 |
| 1.1. | Iniciar sesión | 3 |
| 1.2. | El <i>Notebook</i> de Jupyter | 5 |
| 1.2.1. | Sistema de archivos | 5 |
| 1.2.2. | Hojas de trabajo | 7 |
| 1.2.3. | Interacción con el sistema operativo | 8 |
| 1.3. | Notación | 9 |
| 1.4. | En resumen | 10 |
| A | Uso de este documento | 12 |
| B | L^AT_EX básico | 15 |
| 2 | SAGE como calculadora avanzada | 21 |
| 2.1. | Aritmética elemental | 22 |
| 2.2. | Listas y tuplas | 24 |
| 2.3. | Funciones | 25 |
| 2.3.1. | Variables y expresiones simbólicas | 26 |
| 2.3.2. | Variables booleanas | 30 |

| | | |
|----------|---|-----------|
| 2.4. | Gráficas | 30 |
| 2.4.1. | Gráficas en 2D | 30 |
| 2.4.2. | Gráficas en 3D | 31 |
| 2.4.3. | Gráficas 3D en el notebook antiguo | 34 |
| 2.5. | Cálculo | 34 |
| 2.5.1. | Ecuaciones | 35 |
| 2.5.2. | Límites | 36 |
| 2.5.3. | Series | 37 |
| 2.5.4. | Cálculo diferencial | 37 |
| 2.5.5. | Desarrollo de Taylor | 38 |
| 2.5.6. | Cálculo integral | 38 |
| 2.6. | Álgebra lineal | 39 |
| 2.6.1. | Construcción de matrices | 40 |
| 2.6.2. | Submatrices | 41 |
| 2.6.3. | Operaciones con matrices | 42 |
| 2.6.4. | Espacios vectoriales | 43 |
| 2.6.5. | Subespacios vectoriales | 44 |
| 2.6.6. | Bases y coordenadas | 45 |
| 2.6.7. | Producto escalar | 45 |
| 2.7. | Ejercicios | 46 |
| 2.7.1. | Inducción y sucesiones | 46 |
| 2.7.2. | Un ejercicio de cálculo | 48 |
| 2.7.3. | Algunos ejercicios más | 50 |
| 2.7.4. | Ejercicios de Álgebra Lineal | 51 |
| 3 | Estructuras de datos | 54 |
| 3.1. | Datos básicos: tipos | 55 |
| 3.2. | Listas | 56 |
| 3.2.1. | Métodos y funciones con listas | 58 |
| 3.2.2. | Otras listas | 59 |
| 3.2.3. | Ejercicios con listas | 61 |

| | |
|---|-----|
| ÍNDICE GENERAL | III |
| 3.3. Tuplas | 61 |
| 3.3.1. Ejemplos con tuplas | 63 |
| 3.4. Cadenas de caracteres | 64 |
| 3.4.1. Ejercicios | 67 |
| 3.5. Conjuntos | 67 |
| 3.5.1. Ejercicios | 69 |
| 3.6. Diccionarios | 70 |
| 3.7. Conversiones | 72 |
| 4 Técnicas de programación | 75 |
| 4.1. Funciones | 75 |
| 4.2. Control del flujo | 77 |
| 4.2.1. Bucles for | 77 |
| 4.2.2. Contadores y acumuladores | 80 |
| 4.2.3. Otra sintaxis para los bucles | 81 |
| 4.2.4. Otra más | 82 |
| 4.2.5. Los bloques if <condicion>: | 83 |
| 4.2.6. Bucles while | 84 |
| 4.3. Recursión | 87 |
| 4.4. Depurando código | 90 |
| 4.5. Ejemplos | 92 |
| 4.5.1. Órbitas | 92 |
| 4.5.2. Ordenación | 95 |
| 4.6. Ejercicios | 98 |
| C Máquina virtual | 103 |
| D Tortuga | 108 |
| 5 Complementos | 113 |
| 5.1. Sistemas de numeración | 114 |

| | | |
|--------|--|-----|
| 5.1.1. | Cambios de base | 114 |
| 5.1.2. | Sistemas de numeración en Sage | 115 |
| 5.1.3. | La prueba del 9 y otros trucos similares | 116 |
| 5.2. | Trucos | 116 |
| 5.2.1. | Potencias | 117 |
| 5.2.2. | Listas binarias | 119 |
| 5.2.3. | Archivos binarios | 120 |
| 5.2.4. | La Enciclopedia de sucesiones de enteros | 121 |
| 5.2.5. | Enlaces a otras zonas del documento | 121 |
| 5.3. | Fuerza bruta | 122 |
| 5.4. | Cálculo en paralelo | 123 |
| 5.5. | Eficiencia | 125 |
| 5.5.1. | Control del tiempo | 126 |
| 5.5.2. | Control de la RAM | 127 |
| 5.5.3. | Cython | 128 |
| 5.5.4. | Numpy | 129 |

II Aplicaciones

131

| | | |
|--------|---|-----|
| 6 | Teoría de números | 132 |
| 6.1. | Grupos, anillos y cuerpos | 133 |
| 6.1.1. | Grupos | 133 |
| 6.1.2. | Anillos | 134 |
| 6.1.3. | Cuerpos | 135 |
| 6.2. | Clase de restos | 135 |
| 6.2.1. | Teorema de Fermat-Euler | 136 |
| 6.3. | Fibonacci | 138 |
| 6.3.1. | Algunas propiedades de la sucesión de Fibonacci | 140 |
| 6.4. | Algoritmo de Euclides | 141 |
| 6.4.1. | Mínimo común múltiplo | 143 |

| | | |
|----------|---|------------|
| 6.4.2. | MCD en Sage | 143 |
| 6.5. | Números primos | 144 |
| 6.5.1. | Números primos en Sage | 145 |
| 6.5.2. | Algunos problemas | 146 |
| 6.6. | Enteros representables como sumas de cuadrados | 148 |
| 6.7. | Desarrollos decimales | 148 |
| 6.8. | Ejercicios | 150 |
| 7 | Aproximación | 154 |
| 7.1. | Precisión | 156 |
| 7.2. | El número π | 156 |
| 7.2.1. | Arquímedes (S. III a.C.) | 159 |
| 7.2.2. | Leonhard Euler (1707-1783) | 162 |
| 7.2.3. | Srinivasa Ramanujan (1914) | 163 |
| 7.2.4. | Salamin y Brent (1976) | 163 |
| 7.2.5. | Chudnovsky (199*) | 164 |
| 7.2.6. | Rabinowitz y Wagon (1995) | 167 |
| 7.2.7. | ¿Cuál es la cifra que ocupa el lugar 1000001 en π ? (BBP) | 167 |
| 7.2.8. | Aproximaciones racionales de π | 170 |
| 7.3. | Fracciones continuas | 172 |
| 7.3.1. | Las fracciones continuas aparecen al dividir | 172 |
| 7.3.2. | Las fracciones continuas aparecen al resolver ecuaciones | 173 |
| 7.3.3. | Fracciones continuas infinitas | 173 |
| 7.3.4. | Algoritmo canónico | 175 |
| 7.4. | Fracciones continuas periódicas | 178 |
| 7.4.1. | Ecuaciones de Pell | 182 |
| 7.5. | Logaritmos | 185 |
| 7.5.1. | Un problema de Gelfand | 186 |
| 7.6. | Ceros de funciones | 187 |
| 7.6.1. | Existencia de logaritmos | 187 |
| 7.6.2. | Método de Newton | 189 |

| | | |
|----------|---|------------|
| 7.6.3. | Bisección | 190 |
| 7.7. | Aproximación de funciones | 190 |
| 7.7.1. | Interpolación de Lagrange | 191 |
| 7.7.2. | Interpolación de Newton y polinomio de Taylor | 191 |
| 7.7.3. | <code>find_fit</code> | 193 |
| 7.7.4. | Otros <code>find_...</code> | 194 |
| 7.7.5. | Aproximación global | 194 |
| 7.8. | Ejercicios | 196 |
| 8 | Criptografía | 202 |
| 8.1. | Codificación | 203 |
| 8.2. | Criptoanálisis | 204 |
| 8.3. | Criptografía clásica | 204 |
| 8.3.1. | Cifra de César | 204 |
| 8.3.2. | Cifrado de permutación | 205 |
| 8.3.3. | Cifra de Vigenere | 206 |
| 8.3.4. | Cifrado matricial | 207 |
| 8.3.5. | One time pad | 208 |
| 8.3.6. | En resumen | 210 |
| 8.4. | Clave pública: RSA | 210 |
| 8.5. | Otro sistema de clave pública: MH | 213 |
| 8.6. | Firma digital | 215 |
| 8.7. | Ejercicios | 217 |
| 9 | Más teoría de números | 222 |
| 9.1. | ¿Cómo encontrar primos muy grandes? Miller-Rabin (1980) | 223 |
| 9.1.1. | AKS | 224 |
| 9.2. | ¿Cómo factorizar un entero grande? | 225 |
| 9.2.1. | Fermat | 225 |
| 9.2.2. | Pollard $p - 1$ (1974) | 226 |
| 9.3. | Irreducibilidad y factorización de polinomios | 227 |

| | | |
|-----------|--|------------|
| 9.3.1. | Polinomios con coeficientes racionales | 228 |
| 9.3.2. | Polinomios con coeficientes enteros | 229 |
| 9.3.3. | Polinomios en Sage | 229 |
| 9.3.4. | Factorización de Kronecker | 230 |
| 9.3.5. | Criterio de irreducibilidad de Brillhart | 231 |
| 10 | Matemática discreta | 233 |
| 10.1. | Permutaciones, variaciones y combinaciones | 234 |
| 10.2. | Funciones entre conjuntos finitos | 235 |
| 10.3. | Acciones de grupos | 236 |
| 10.4. | Grafos | 237 |
| 10.4.1. | Grafos en Sage | 239 |
| 11 | Probabilidad | 243 |
| 11.1. | Probabilidad | 244 |
| 11.1.1. | Variables aleatorias | 246 |
| 11.1.2. | Simulación | 246 |
| 11.2. | Binomial | 249 |
| 11.2.1. | Caso general | 250 |
| 11.3. | Generadores de números (pseudo-)aleatorios | 251 |
| 11.3.1. | Generación eficiente de números aleatorios | 252 |
| 11.4. | Elementos aleatorios | 253 |
| 11.4.1. | Grafos aleatorios | 253 |
| 11.5. | Simulación: método de Monte Carlo | 254 |
| 11.5.1. | Cálculo aproximado de π | 254 |
| 11.5.2. | Cálculo de áreas | 255 |
| 11.5.3. | Cálculo de integrales | 255 |
| 11.5.4. | Cálculo de probabilidades | 256 |
| 11.5.5. | Un sistema físico | 259 |
| 11.5.6. | Cadenas de Markov | 260 |
| 11.5.7. | Paseos aleatorios | 261 |

| | |
|---|------------|
| 11.5.8. Urnas de Polya | 262 |
| 11.5.9. Probabilidades geométicas | 263 |
| 11.6. Ejercicios | 264 |
| 12 Miscelánea | 266 |
| 12.1. Autómatas celulares | 266 |
| 12.1.1. El Juego de la Vida | 267 |
| 12.1.2. Autómatas celulares 1-dimensionales | 269 |
| 12.2. Percolación | 270 |
| 12.3. Ley de Benford | 272 |
| 12.4. Tratamiento de imágenes | 273 |
| 12.5. Sistemas polinomiales de ecuaciones | 274 |
| 12.5.1. Bases de Gröbner en Sage | 275 |
| 12.6. Ejercicios | 277 |
| E Recursos | 281 |
| E.1. Bibliografía | 281 |
| E.2. Otros | 283 |

Prólogo

La idea detrás de la existencia de un curso como este es fácil de expresar: *es muy conveniente que alumnos de un Grado en Ciencias Matemáticas tengan la oportunidad, desde el comienzo de sus estudios, de conocer y practicar una aproximación experimental al estudio de las matemáticas.*

La idea de las matemáticas como una ciencia experimental aparece después de que se generalizara el uso de los ordenadores personales, en nuestro país a partir de los años 90, y, sobre todo, con la aparición de ordenadores personales con suficiente potencia de cálculo ya en este siglo.

Sin embargo, algunos matemáticos de siglos anteriores demostraron una extraordinaria capacidad de cálculo mental y realizaron sin duda experimentos que les convencieron de que las ideas que pudieran tener sobre un determinado problema eran correctas, o bien de que eran incorrectas. Un ejemplo claro es el de Gauss en sus trabajos sobre teoría de números: se sabe, ya que se conservan los cuadernos que utilizaba, que fundamentaba sus afirmaciones en cálculos muy extensos de ejemplos concretos.

Los ordenadores personales extienden nuestra capacidad de cálculo, en estos tiempos bastante limitada, y permiten realizar búsquedas exhaustivas de ejemplos o contraejemplos. Podemos decir, hablando en general, que los ordenadores realizan para nosotros, de forma muy eficiente, tareas repetitivas como la ejecución, miles de veces, de un bloque de instrucciones que por algún motivo nos interesa.

Es cierto que el ordenador no va a demostrar un teorema por nosotros, pero puede encontrar un contraejemplo, probando así que el resultado propuesto es falso, y también puede, en ocasiones, ayudarnos a lo largo del desarrollo de una demostración. Sobre todo nos convencer de que una cierta afirmación es plausible y, por tanto, puede merecer la pena pensar sobre ella.

DESCRIPCIÓN DEL CURSO

En este curso usamos Sage como instrumento para calcular y programar. Puedes ver una descripción de la forma de acceder a su uso en el primer capítulo de estas notas.

Comenzamos estudiando el uso de Sage como calculadora avanzada, es decir, su uso para obtener respuesta inmediata a nuestras preguntas mediante programas ya incluidos dentro del sistema. Por ejemplo, mediante `factor(n)` podemos obtener la descomposición como producto de factores primos de un entero `n`.

Sage tiene miles de tales instrucciones preprogramadas que resuelven muchos de los problemas con los que nos podemos encontrar. En particular, revisaremos las instrucciones para resolver problemas de cálculo y álgebra lineal, así como las instrucciones para obtener representaciones gráficas en 2 y 3 dimensiones.

A continuación, después describir las estructuras de datos y su manipulación, de trataremos los aspectos básicos de la programación usando el lenguaje Python, que es el lenguaje utilizado en una gran parte de Sage, y es el que usaremos a lo largo del curso. Concretamente, veremos los bucles `for` y `while`, el control del flujo usando `if` y `else`, y la recursión. Con estos pocos mimbres haremos todos los cestos que podamos durante este curso.

Por último, el curso contiene cuatro bloques, aproximación, aritmética, criptografía y teoría de la probabilidad, que desarrollamos todos los cursos junto con uno más, que llamamos miscelánea, cuyo contenido puede variar de unos cursos a otros. En esta parte usamos los rudimentos de programación que hemos visto antes para resolver problemas concretos dentro de estas áreas.

Aunque puede parecer que no hay conexión entre estos bloques, veremos que la hay bastante fuerte:

1. La aritmética, el estudio de los números enteros, es la base de muchos de los sistemas criptográficos que usamos para transmitir información de forma segura. En particular, estudiaremos el sistema RSA, uno de los más utilizados actualmente, cuya seguridad se basa en la enorme dificultad de factorizar un entero muy grande cuyos únicos factores son dos números primos enormes y distantes entre sí.
2. Para elegir los factores primos en el sistema RSA, cada usuario del sistema debe tener su par de primos, se utilizan generadores de números (pseudo-)aleatorios. Este será uno de los asuntos que trataremos en el bloque de probabilidad.
3. En el capítulo de ampliación de teoría de números discutiremos cómo encontrar números primos muy grandes y cómo intentar factorizar, de manera eficiente, números grandes. Son dos asuntos muy relacionados con la criptografía.
4. En el bloque sobre la aproximación de números reales, dedicaremos cierta atención al cálculo de los dígitos de algunas constantes matemáticas, en particular π y e . También estudiaremos la resolución aproximada de ecuaciones y la aproximación de funciones.

5. Como aplicación de este bloque sobre aproximación de números reales veremos un par de ejemplos en que usaremos logaritmos para estudiar potencias a^n con n muy grande. Es decir, usaremos los reales (*el continuo*) para estudiar un problema discreto.

De la lista anterior se deduce que el tema central del curso es la criptografía, con varios de los otros temas ayudando a entender y aplicar correctamente los sistemas criptográficos.

Los temas que tratamos en la segunda parte del curso vuelven a aparecer en asignaturas de la carrera, en algún caso optativas como la criptografía, y pueden verse como pequeñas introducciones *experimentales* a ellos. Creemos entonces que la asignatura es una buena muestra, por supuesto incompleta, de lo que os encontraréis durante los próximos años. Es como catar *el melón* antes de abrirlo.

El prototipo de este curso fué desarrollado, en su totalidad, por Pablo Angulo, y puedes todavía consultar el excelente resultado de su trabajo en este [enlace](#). Los que lo hemos impartido después aprendimos casi todo lo que sabemos en sus notas y, por supuesto, se lo agradecemos aquí.

Parte I

Fundamentos

Capítulo 1

Introducción

Sage es un sistema de álgebra computacional (CAS, del inglés *computer algebra system*). El programa es libre, lo que nos permite copiarlo, modificarlo y redistribuirlo libremente. Sage consta de un buen número de librerías para ejecutar cálculos matemáticos y para generar gráficas. Para llamar a estas librerías se usa el lenguaje de programación **Python**.

Sage está desarrollado por el proyecto de software libre [Sagemath](#). Se encuentra disponible para GNU/Linux y MacOS, y para Windows bajo máquina virtual (*virtualbox*)¹. Reúne y compatibiliza bajo una única interfaz y un único entorno, distintos sistemas algebraicos de software libre.

Python es un lenguaje de propósito general de muy alto nivel, que permite representar conceptos abstractos de forma natural y, en general, hacer más con menos código. Buena parte de las librerías que componen Sage se pueden usar directamente desde Python, sin necesidad de acarrear todo el entorno de Sage.

Existen varias formas de interactuar con Sage: desde la consola, desde ciertos programas como TeXmacs o Cantor, y desde el *navegador de internet*. Para este último uso, Sage crea un *servidor web* que escucha las peticiones del cliente (un navegador), realiza los cálculos que le pide el cliente, y le devuelve los resultados. En esta asignatura sólo usaremos el interfaz web (*notebook*).

¹Recientemente se ha presentado una versión de Sage para Windows que no requiere la máquina virtual, es decir, puede instalarse directamente. Pueden verse los detalles en [esta página](#).

1.1. Iniciar sesión

1. EN UNA MÁQUINA LOCAL (no se necesita conexión a internet)

- a) de nuestro laboratorio: abrir una terminal (la encontramos en el menú **Aplicaciones/Herramientas del Sistema/Terminal** de **MATE**) y ejecutar (escribir el texto indicado y pulsar **Intro**)

```
arrancar—jupyter.sh
```

Lo único que hace este **script** es cambiar de directorio a

```
/Desktop/SAGE-noteb/IPYNB/
```

y ejecutar **sage -notebook=jupyter --ip=127.0.0.1 --port=8888** que arranca Sage con la interfaz de **Jupyter**. Si en la terminal ejecutamos **sage -notebook=sagenb** accederíamos al **notebook** antiguo de Sage. En este curso usaremos siempre el nuevo, es decir **Jupyter**.

- b) exterior a nuestro laboratorio: generalmente se ha de instalar previamente el software. En el sitio de [Sagemath](#) se encontrarán las instrucciones precisas para cada sistema operativo². Una vez instalado, el inicio de sesión será similar al indicado en el punto anterior.

En este caso el programa se ejecuta en la máquina en la que estamos sentados (*máquina local=localhost*) y el navegador web se conecta a la máquina local mediante la dirección

<http://localhost:8888/>.

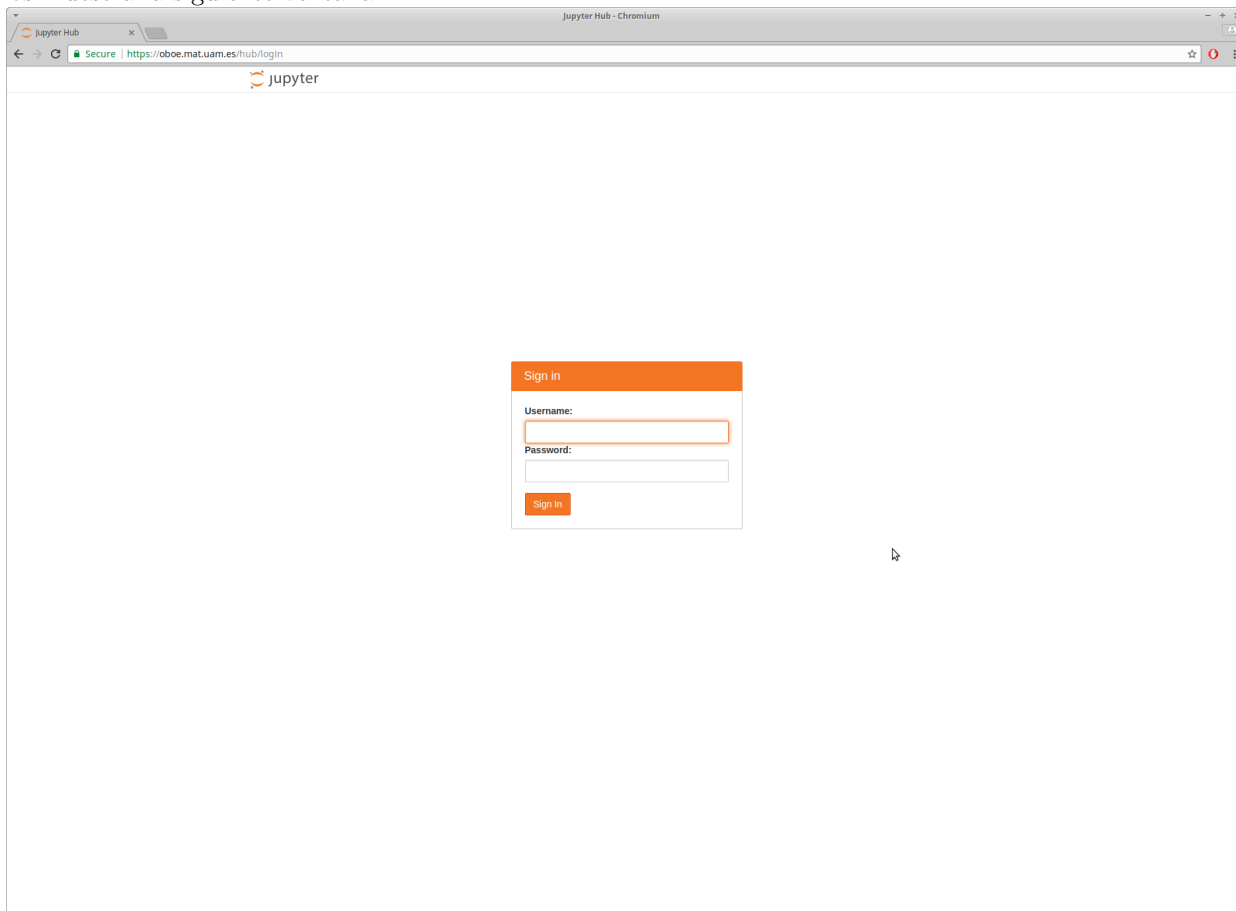
El número del puerto, por defecto 8888, se puede utilizar si no hay otro proceso en la máquina que lo haya reservado. Si el puerto está ocupado **Jupyter** utiliza otro y los enlaces a hojas (Apéndice **A-4c**) de Sage no funcionarán. Si se trata, como ocurrirá casi siempre, de que ya hemos arrancado **Jupyter** en nuestra sesión, podemos ejecutar en una terminal **killall python** para poder arrancar **Jupyter** en el puerto 8888.

2. EN EL SERVIDOR DEL DEPARTAMENTO DE MATEMÁTICAS (se necesita conexión a internet). Al teclear, en un navegador, la dirección

<https://jupyter.mat.uam.es/>,

²En este momento todavía no aparecen las instrucciones para la instalación en Windows sin máquina virtual mencionada en la nota anterior.

se nos muestra la siguiente ventana



The image shows a web browser window titled "Jupyter Hub - Chromium". The address bar displays "Secure | https://oboe.mat.uam.es/hub/login". The page features the Jupyter logo and a "Sign in" form. The form has an orange header bar with the text "Sign in". Below this, there are two input fields: "Username:" and "Password:". The "Username:" field is highlighted with an orange border. At the bottom of the form is an orange "Sign In" button. A mouse cursor is visible near the bottom right of the form.

Sign in

Username:

Password:

Sign In

El usuario es del tipo **nombre-apellido-jup**, nombre y apellido como en vuestra dirección de correo de la UAM, y

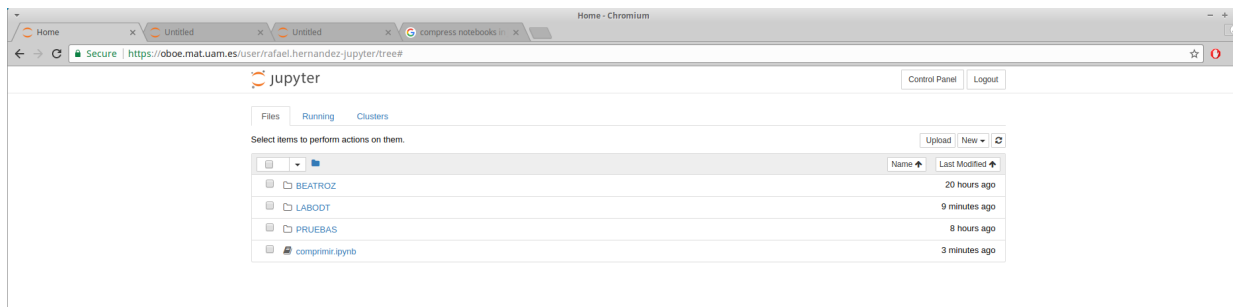
contraseña que debe ser la misma que se os da para las cuentas en el Laboratorio.

Cuando conectamos el navegador a una dirección remota, es decir, no a `localhost`, el código de Sage se ejecuta en la máquina remota y la máquina en la que estemos trabajando únicamente ejecutará el navegador. Si el servidor tiene que atender peticiones de cálculos complicados de muchos usuarios simultáneamente puede ralentizarse su funcionamiento, o incluso puede colgarse. En consecuencia, para trabajar durante las clases debería usarse la máquina local, como explicado en 1.1-1a, en lugar del servidor del Departamento.

1.2. El *Notebook* de Jupyter

Una vez que conectamos, a través del navegador, con una sesión de **Jupyter**, accedemos a una página en la que podemos ver el que va a ser nuestro *espacio de trabajo* básico.

1.2.1. Sistema de archivos



1. Si estamos trabajando en el Laboratorio, y por tanto hemos ejecutado `arrancar-jupyter.sh` en una terminal para iniciar la sesión, las carpetas y archivos que vemos son los que hay en el directorio `~/Desktop/SAGE-noteb/` (el símbolo `~` representa el directorio `HOME` del usuario), que es el que contiene los archivos que os damos para el curso (ver Apéndice A-1).

La terminal desde la que arrancamos queda abierta y en ella van apareciendo mensajes que se refieren a la ejecución de **Jupyter**. Esta terminal se puede mandar al panel, pinchando en el pequeño guión que hay en la parte de arriba a la

derecha de la ventana, pero debe permanecer viva.

Al terminar de trabajar, es IMPORTANTE cerrar bien `jupyter` pulsando `control-c` con la ventana de la terminal activa. Es decir, la buscamos en el panel, pinchamos en ella para activarla y entonces pulsamos las teclas `control` y `c` al mismo tiempo dos veces seguidas.

2. Si, en cambio, hemos iniciado sesión en el servidor del Departamento, las carpetas y archivos que vemos son las que teníamos en el servidor, tal como quedaron en la sesión anterior.

Cuando terminamos de trabajar, pinchamos en la pestaña `Running` para ver las hojas que todavía están activas, aparecen en color verde, y las paramos todas marcándolas en la casilla de la izquierda y pulsando `Shutdown`. Finalmente, pinchamos en la pestaña `Files` y en la línea superior de la ventana pinchamos en `Logout`.

Aquí también es IMPORTANTE cerrar, como se acaba de indicar, los procesos en el servidor que nos han permitido el acceso.

El resto de las indicaciones se aplican a los dos casos: Laboratorio y servidor del Departamento.

1. Cuando pinchamos en el cuadrado a la izquierda de un nombre de archivo se activan cuatro nuevas posibilidades: *Duplicate*, *Rename*, *Move*, *Download* que no necesitan grandes explicaciones. *Move* permite mover el archivo marcado a cualquiera de las carpetas que existen en nuestro directorio.

En el menú *New* se puede crear una carpeta nueva dentro de la que está activa en ese momento, y podemos navegar en el sistema de archivos pinchando en los nombres de las carpetas.

2. Al seleccionar un archivo o carpeta también aparece un icono de papelera de color rojo. Permite destruirlos y, aunque pide confirmación, es BASTANTE PELIGROSO ya que puede haber archivos o carpetas seleccionados pero que no vemos en la ventana del navegador. Debemos usarlo poco y asegurarnos de que únicamente estamos destruyendo lo que realmente queremos.

Una opción razonable es crear una carpeta de nombre `BASURA` en la página de inicio y mover los archivos a ella en lugar de destruirlos. Al cabo de un tiempo deberemos limpiar esta carpeta de archivos que realmente no son necesarios.

3. A la derecha de la ventana, encima de la lista de archivos y carpetas, también hay un botón *Upload* para subir archivos desde el sistema de archivos del ordenador en el que estamos trabajando al sistema de archivos del servidor del Departamento.

En el caso de que estemos trabajando en *local*, este botón NO ES NECESARIO ya que podemos ver, desde **Jupyter**, todos los archivos que hay por debajo de la carpeta en la que hemos arrancado (1.2.1-1).

4. Cuando marcamos dos o más archivos desaparece la posibilidad de descargarlos *clizando* una sólo vez. Hay que descargarlos uno a uno, o bien comprimirlos en un sólo archivo antes de descargarlo. De la misma forma, se puede comprimir una carpeta con archivos, que queremos subir al servidor, para poder subirlos todos de golpe. Se explica la forma más adelante (1.2.3).
5. Debe estar claro que conviene usar estas funciones de la interfaz de **Jupyter** para mantener nuestro ESPACIO DE TRABAJO ORDENADO. La forma concreta de hacerlo dependerá de cada persona.

1.2.2. Hojas de trabajo

1. Creamos una hoja de trabajo vacía en el menú *New* eligiendo el tipo de hoja en el desplegable. En el Laboratorio los únicos tipos disponibles serán **Python** y **Sagemath**, pero en el servidor del Departamento hay algunos más: **R** para Estadística, **Matlab**, **Julia** y **Octave** para Cálculo Numérico, **C++11** para programar, de forma interactiva, en **C++**.
2. Cada uno de estos tipos de hoja está asociado a un **núcleo (kernel)** que ejecuta los cálculos que le indicamos. En ocasiones hay que interrumpir la ejecución o reiniciar el **núcleo**. Cuando se reinicia se pierden todas las funciones y variables que teníamos.

Todo esto se hace bajo el menú desplegable **Kernel**. La última entrada del menú sirve para cambiar de núcleo, y que usaremos (raramente) cuando una hoja de Sage sea entendida por el sistema como si fuera de Python.

3. Las celdas que vemos en la hoja son, por defecto, **celdas de cálculo**. Es decir, en ellas escribimos el código que queremos ejecutar. Encima de la zona de celdas hay un menú con pequeños iconos, y al pasar el cursor por encima de ellos se autoexplican. En particular, el triangulito con el segmento vertical en un vértice sirve para ejecutar la celda activa y el cuadrado para parar la ejecución. La celda activa tiene su borde de color verde.

También es posible ejecutar la celda activa pulsando **control+Intro** (las dos teclas al mismo tiempo).

4. En ocasiones es necesario reiniciar totalmente una hoja que se ha colgado. Se puede utilizar el último, de izquierda a derecha, de los iconos mencionados en el punto anterior, que muestra una flecha circular. Al hacer esto se borran de la memoria todos los cálculos anteriores y hay que volver a ejecutar las celdas que nos convenga.

5. En el submenú **File/Download** as podemos elegir **PDF via Latex (.pdf)** para convertir la hoja a formato PDF, por ejemplo para imprimirla bien formateada. Las líneas de código que son demasiado largas y salen de la página, en el PDF, se pueden cortar usando el carácter `'\'` para terminarlas, es decir, varias líneas consecutivas terminadas cada una, menos la última, en `'\'` se interpretan como una única línea.
6. En las celdas de cálculo, en hojas **Sagemath**, hay un sistema para completar comandos pulsando el tabulador, y un sistema completo de ayuda que se obtiene completando el nombre de un comando con un interrogante (?) y ejecutando la celda.
7. Convertimos una **celda** de cálculo en una de **texto** en el submenú *Cell/Cell Type*, eligiendo como tipo *Markdown*.
En estas celdas se puede escribir código L^AT_EX(ver Capítulo 1-Apéndice B) entre dólares, para las fórmulas matemáticas, y usar el lenguaje **markdown** para formatear el texto normal: secciones, subsecciones, listas, etc. Las celdas de texto también hay que ejecutarlas, en la misma forma que las de cálculo, para ver el resultado.
Puedes ver un resumen del uso de *Markdown* en [esta hoja](#)³.
8. Es fácil convertir las hojas de trabajo antiguas, con extensión **.sws** a hojas nuevas (hojas de **Jupyter**) con extensión **.ipynb**. Las instrucciones y programas necesarios están en la carpeta **SAGE-noteb/bin/sws2ipynb/** en tu escritorio.
9. Hay otras opciones en la interfaz de **Jupyter** pero, aunque quizá útiles, no es muy probable que las usemos.

1.2.3. Interacción con el sistema operativo

1. Una hoja de trabajo de tipo **Python** permite interaccionar con el sistema operativo de la máquina. En el caso de que estemos trabajando en **local** no es muy útil porque podemos hacer lo mismo en una terminal, pero en el caso de que estemos trabajando en **remoto** sí lo es.
2. En una celda de cálculo, de una hoja de **Python**, se pueden escribir comandos del sistema operativo sin más que comenzar la línea con el símbolo de admiración (!). El comportamiento que se obtiene es el mismo que en una terminal, y depende de los permisos de ejecución que tenga el usuario.

Puedes ver ejemplos de esta interacción, por ejemplo comprimir y descomprimir archivos, en [esta otra hoja](#).

³El enlace funciona si se ha arrancado **Jupyter** como se indica en 1.1-1a.

3. Esta posibilidad es intrínsecamente peligrosa ya que cualquier pequeño error en la configuración del sistema podría permitir actividades no deseadas, como, por ejemplo, borrar el disco duro.

En el caso en que encontréis un error de este tipo, en las máquinas del Laboratorio o en el servidor, rogamos que lo comunicuéis cuanto antes a la dirección de correo

`informatica.matematicas@uam.es`.

1.3. Notación

En estas notas, representaremos los cuadros de código por una

caja con fondo de color .

En ocasiones aparecerán numeradas las líneas de código, en la parte exterior de la caja. Esta numeración no es parte del código y aparece para facilitar la referencia a líneas individuales. Además, las palabras clave del lenguaje de programación aparecerán resaltadas, para distinguirlas de las demás. Las respuestas del intérprete, en caso de querer mostrarlas, aparecerán indentadas, y en otro color, bajo las cajas con el código.

Así, por ejemplo, transcribiremos la celda

```
## Suma de los 150 primeros impares positivos
m=150
'Los %d primeros impares positivos suman %d.'%(m,sum([2*k+1 for k in range(m)]))
'Los 150 primeros impares positivos suman 22500.'

'Los 150 primeros impares positivos suman 22500.'
```

con alguno de los siguientes aspectos

- o numerado

```
1 ## Suma de los 150 primeros impares positivos
2 m=150
3 'Los %d primeros impares positivos suman %d.'%(m,sum([2*k+1 for k in range(m)]))
```

- o sin numerar

```
## Suma de los 150 primeros impares positivos
m=150
'Los %d primeros impares positivos suman %d.' %(m,sum([2*k+1 for k in range(m)]))
```

- o con la respuesta del intérprete

```
## Suma de los 150 primeros impares positivos
m=150
'Los %d primeros impares positivos suman %d.' %(m,sum([2*k+1 for k in range(m)]))

'Los 150 primeros impares positivos suman 22500.'
```

Cuando copiamos código desde este PDF a una celda de Sage podemos encontrarnos con errores de sintaxis debidos simplemente a que en la celda se entienden algunos caracteres, que estaban en el PDF, de forma incompatible con las reglas sintácticas de Sage. Un ejemplo típico aparece con el guión que usamos para la resta, que al pegarlo en la celda aparece como un guión largo, que Sage no interpreta como el símbolo de la resta.

1.4. En resumen

Cuando trabajamos en **local** los recursos de la máquina, **RAM**, **cores**, **red**, **etc.**, son nuestros, pero ese no es el caso al trabajar con el servidor de cálculo del Departamento, ya que estamos compartiendo los recursos con el resto de los usuarios.

1. El servidor dispone de 500 *GB* para almacenar los archivos de trabajo de los usuarios. Aunque no se ha fijado, de momento, un límite de espacio por usuario esperamos que ningún usuario supere unos 2 *GB* de espacio en disco.

En el caso de las máquinas del Laboratorio también debe haber límites, ya que todos los archivos de todos los usuarios se almacenan en un único disco duro de 256 *GB*, lo que nos permite sentarnos en cualquier puesto y ver los archivos de nuestra cuenta. En este caso un límite razonable puede ser de 1 *GB*.

2. En el servidor hay un límite de 3 *GB* de **RAM** por usuario conectado. Es NECESARIO cerrar las hojas que no estemos usando, para trabajar basta con tener una hoja abierta y puntualmente abrir otra para copiar algún código, y no dejar procesos vivos en la máquina, como se explica en la página 6, cuando hemos dejado de trabajar.
3. En este momento todavía no sabemos cuántos usuarios simultáneos soporta, sin ralentizarse, el servidor. Entonces, no parece conveniente usar el servidor durante las clases ya que podría colapsarse y hacernos perder tiempo.

4. Aunque se hace un *backup* cada noche del servidor de cálculo y de las cuentas del Laboratorio, no deberíais fiaros completamente de esto. Es conveniente mantener un *backup* personal en un lápiz de memoria.

Apéndice A

Uso de este documento

Hemos preparado estas notas con la intención de que faciliten el mantenimiento organizado de toda la información que genera el curso. Pueden cambiar un poco a lo largo del curso, ya que corregimos las erratas, y errores, que se detecten, y añadiremos nuevas secciones o temas si nos parece útil. Por otra parte, también pretendemos que sigan creciendo en cursos sucesivos aunque entonces será probablemente imposible cubrir todo el material y habrá que seleccionar.

1. Encontrarás una carpeta **SAGE-noteb** en el escritorio de tu cuenta en el Laboratorio. Esta carpeta contendrá los materiales que os vayamos dando, y, por tanto, su contenido puede variar de una semana a otra. Se recomienda mantener una copia actualizada de esta carpeta en un *pendrive*.
2. En cada examen encontrarás una copia de la carpeta **SAGE-noteb**, tal como estaba en tu cuenta habitual justo antes del examen, en el escritorio de la cuenta en la que debes hacer el examen. Esto quiere decir que puedes colocar archivos que quieras ver durante el examen en ciertas subcarpetas, se concreta un poco más adelante, de la carpeta **SAGE-noteb**.
3. El documento básico es este, **laboratorio.pdf**, que tiene un montón de enlaces a otras páginas del documento, a páginas *web* y a otros documentos, lecturas opcionales, que están situados en la carpeta **PDFs** dentro de la que contiene todo el material. Se trata entonces de un documento *navegable*.

RECOMENDAMOS abrirlo con el navegador *web*, en nuestro caso usamos **chromium-browser** por defecto. Se abre escribiendo en la barra de direcciones del navegador


```
file:///alumnos/<curso>/<usuario>/Desktop/SAGE-noteb/laboratorio.pdf
```

con `<curso>` el nombre que tiene en el Laboratorio (`labodt`, `labot`, etc.) y `<usuario>` el de vuestra cuenta en el Laboratorio.

4. ENLACES: navegamos en el documento y fuera de él usando diversos tipos de enlaces:

- a) Por ejemplo, este es un [enlace a un documento PDF](#) situado en uno de nuestros servidores y debe abrirse en el navegador que usamos por defecto, mientras que este otro es un [enlace a una página web](#) externa.
- b) También hay [enlaces a otras zonas de este mismo documento](#). Debemos recordar, aproximadamente, la página del documento en la que estábamos para poder volver cómodamente a ella. Puede ayudar recordar la zona de la barra de la derecha de la ventana (barra de *scroll*) en la que estaba la página de origen ya que *clickando* en esa zona volvemos a ella.
- c) Por último, hay [enlaces](#) que nos llevan directamente a hojas de trabajo de Sage que, como hemos visto se abren dentro del navegador (en nuestro caso **Chromium**). Para que estos enlaces funcionen hay que arrancar Sage como se indica en [1.1-1a](#). Además conviene abrir esos enlaces en pestañas nuevas del navegador *clickando* en el enlace con el botón medio, es decir, con la rueda. Alternativamente, se puede *clickar* en el enlace con el botón derecho y seleccionar en el menú que aparece “*Open link in new tab*”.

5. Puedes añadir tus propias notas para completar o clarificar el contenido de nuestro documento. Es importante entonces tener en cuenta que puedes, y debes, *personalizar* nuestro `laboratorio.pdf` con tus aportaciones o las de tus compañeros. Para esto

- a) Para cada capítulo, por ejemplo el 4, hay un documento, en la carpeta `SAGE-noteb/INPUTS/NOTAS`, con nombre `notas-cap4.tex` en el que puedes escribir y no desaparecerá cuando modifiquemos la carpeta. Si escribes en cualquiera de los otros documentos a la semana siguiente puede haber desaparecido lo que hayas añadido.
- b) En esos documentos `notas-capn.tex` se puede escribir texto simple, pero para obtener un mínimo de legibilidad hay que escribir en \LaTeX , que no es sino texto formateado, como se explica en el apéndice siguiente.
- c) Para generar el PDF resultante, incluyendo las notas añadidas, hay que compilar los archivos `tex` (ver Apéndice [B-refTS](#)).

6. Las hojas de trabajo de Sage que hayas creado o modificado y quieras ver durante un examen debes guardarlas en la subcarpeta `IPYNB-mios` dentro de `SAGE-noteb/IPYNB`. Esta subcarpeta puede contener, a su vez, diversas subcarpetas que organicen su contenido.
7. Conviene mantener la información acerca de nuestras hojas de Sage, las que hayamos elaborado o modificado nosotros, de manera que sea fácilmente accesible durante los exámenes: por ejemplo, para una hoja que se refiere al Capítulo 4 de estas notas podrías incluir en el archivo `notas-cap4.tex` un `\item` indicando el nombre y localización del archivo, debería estar en la subcarpeta `IPYNB/IPYNB-mios` de la carpeta principal, y una descripción de su contenido. Esto es importante para facilitar la búsqueda de una hoja concreta sobre la que quizá trabajamos hace cuatro meses y de la que podemos haber olvidado casi todo.
8. En el apéndice **B** se describe la manera de generar enlaces de nuestro PDF, `laboratorio.pdf`, a páginas *web*, a otros documentos PDF o a hojas de trabajo de Sage.

Apéndice B

L^AT_EX básico

1. Aunque este pequeño resumen puede servir para escribir en L^AT_EX las notas que queráis añadir al texto, una introducción bastante completa y clara se puede encontrar en este [enlace](#).
2. Como editor de L^AT_EX usamos el programa **texstudio** que está instalado en las máquinas del Laboratorio.
Una vez que hemos abierto el programa, su lanzador está en el menú *Aplicaciones/Oficina/*, debemos abrir, usando el botón *Open*, los archivos con código L^AT_EX que vamos a editar.
3. Siempre hay que abrir, dentro de **texstudio**, el archivo

SAGE-noteb/laboratorio.tex

que es el documento raíz y el que hay que procesar para obtener el PDF. Se procesa pinchando en el botón **Build&view**, el botón con dos puntas de flecha verdes en la barra superior de **texstudio**. El PDF resultante aparece en el lado derecho de la ventana.

4. Los documentos **notas-capn.tex**, que deberían estar en la subcarpeta **SAGE-noteb/INPUTS/NOTAS/** de la carpeta principal, inicialmente contienen únicamente

```
\begin{enumerate}
  \item
\end{enumerate}
```

que es un entorno de listas numeradas. Debes abrirlos en el editor, usando el menú **O**pen, para añadirles materia.

5. Las líneas, dentro de un documento de código L^AT_EX, que comienzan con un % son comentarios que no aparecen en el PDF resultante. Así, por ejemplo, para ver en el PDF uno de los archivos `notas-capn.tex` que has editado, por ejemplo el `notas-cap2.tex`, debes quitar el símbolo % al comienzo de dos líneas en `SAGE-noteb/laboratorio.tex` cuyo contenido es

```
%\section{Notas personales}
%\montan|notas-cap2|
```

6. Cada nota que quieras incluir debe comenzar con un nuevo `\item` y a continuación el texto que quieras.
7. Para escribir matemáticas dentro de una línea de texto basta escribir el código adecuado entre símbolos de dólar (\$..\$). Para escribir matemáticas en *display*, es decir ocupando las fórmulas toda la línea se puede encerrar el código entre dobles dólares (\$\$. \$\$), o, mucho mejor, abrir la zona de código con `\[` y cerrarla con `\]`.
8. Por ejemplo, podemos mostrar una ecuación cuadrática en *display* mediante

```
\[ax^2+bx+c=0\]
```

que produce

$$ax^2 + bx + c = 0$$

y su solución mediante

```
\[ x=\frac{-b\pm \sqrt{b^2-4ac}}{2a}\]
```

que ahora produce

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

9. Si observas con cuidado el código L^AT_EX anterior verás que la forma en que se escribe el código coincide bastante con la forma en que leemos la expresión. Una diferencia es que ante ciertos operadores con dos argumentos, como la fracción que tiene numerador y denominador, debemos avisar a L^AT_EX de que debe esperar dos argumentos mientras que cuando leemos la fórmula hasta que no llegamos a **partido por 2a** no sabemos que se trata de una fracción.

Esto es lo que hace que aprender a escribir código L^AT_EX sea muy sencillo para personas acostumbradas a leer texto matemático.

10. Para cambiar de párrafo en L^AT_EX basta dejar una línea completamente en blanco.
11. Los subíndices se consiguen con la barra baja, **x_n** da x_n , y los superíndices con el acento circunflejo, **x^n** da x^n .
12. Como se ve en el ejemplo anterior, `\frac{numerador}{denominador}` es la forma de obtener una fracción.
13. Conjuntos:

- a) `$A\times B$` produce $A \times B$.
- b) `$A\cap B$` produce $A \cap B$.
- c) `$A\cup B$` produce $A \cup B$.
- d) `$a\in B$` produce $a \in B$.
- e) `$a\notin B$` produce $a \notin B$.
- f) `$A\subset B$` produce $A \subset B$.
- g) `$A\to B$` produce $A \rightarrow B$.
- h) `$a\mapsto f(a)$` produce $a \mapsto f(a)$.
- i) `$A=\{a,b,c\}$` produce $A = \{a, b, c\}$.

14. Cálculo:

a) `\[\lim_{x\to \infty} f(x)=a\]` produce

$$\lim_{x \rightarrow \infty} f(x) = a.$$

b) `\[\lim_{h\to 0} \frac{f(x+h)-f(x)}{h}=f'(x)\]` produce

$$\lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} =: f'(x).$$

c) `\[\sum_{i=0}^{\infty} \frac{x^n}{n!}=e^x\]` produce

$$\sum_{i=0}^{i=\infty} \frac{x^n}{n!} =: e^x.$$

d) `\[\int_a^b f(x)dx\]` produce

$$\int_a^b f(x)dx.$$

15. También es conveniente saber componer matrices. Por ejemplo,

```
\begin{equation}
\begin{pmatrix}
1&0&0\\
0&1&0\\
0&0&1
\end{pmatrix}
\end{equation}
```

produce la matriz identidad

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{B.1}$$

16. Puedes encontrar una lista más completa de los códigos que producen diversos símbolos matemáticos en este [archivo](#), mientras que la lista completa, que es enorme, se encuentra en [este otro](#).
17. Podemos cambiar el color de un trozo de texto en el PDF sin más que incluir el correspondiente texto, en el archivo con el código \LaTeX , entre llaves indicando el color en la forma `\color{green}...texto...`, que veríamos como `...texto...`
18. Para incluir en el PDF un enlace a otra zona del mismo documento
 - a) En la zona a la que queremos que lleve el enlace debemos incluir una línea con el contenido `\label{nombre}`, donde **nombre** es el nombre arbitrario que damos al enlace y que no debe ser igual a ningún otro *label* en el documento.
 - b) Donde queremos que aparezca el enlace usamos

`\hyperref[nombre]{texto},`

con **nombre** el del enlace de acuerdo al punto anterior, y **texto** el que queramos que aparezca como enlace, es decir coloreado, y que pinchamos para movernos al otro lugar en el documento.

Si incluyes enlaces de estos en la copia de la carpeta **SAGE-noteb** en el ordenador del Laboratorio, y que lleven a zonas del PDF fuera de tus notas personales, *esos enlaces desaparecerán cuando actualicemos* la carpeta.

19. Para incluir en el PDF resultante un enlace a una página *web* basta escribir, en el lugar adecuado del texto, algo como

`\href{http://...URL...}{enlace},`

donde `...URL...` es la dirección completa de la página y **enlace** es el texto que va a aparecer en el PDF como el enlace pinchable.

20. Para incluir en el PDF un enlace a otro PDF, por ejemplo situado en la subcarpeta **PDFs-mios** de la carpeta **SAGE-noteb**, basta escribir, en el lugar adecuado del texto, algo como

`\href{run:PDFs-mios/<nombre del PDF>.pdf}{enlace}.`

El contenido de esta carpeta **PDFs-mios** no desaparecerá al actualizar la carpeta **SAGE-noteb**.

21. En este archivo `laboratorio.pdf` hay también enlaces que llevan directamente a hojas de trabajo de Sage. En las secciones de notas personales puedes crear esa clase de enlaces incluyendo en el código \LaTeX algo como

```
\href{http://localhost:8888/notebooks/<camino>}{texto}
```

con `<camino>` que debería ser

```
~/Desktop/SAGE-noteb/IPYNB/IPYNB-mios/<nombre del archivo>
```

Estos enlaces funcionarán si hemos arrancado Sage como se indica en [1.1-1a](#).

Capítulo 2

SAGE como calculadora avanzada

Usamos Sage como una “calculadora avanzada” escribiendo en una celda de cálculo una instrucción preprogramada junto con un número correcto de parámetros.

Por ejemplo, existe una instrucción para factorizar números enteros, **factor**, que admite como parámetro un número entero, de forma que debemos evaluar una celda que contenga **factor**($2^{137}+1$) para calcular los factores primos del número $2^{137} + 1$.

Muchas instrucciones de SAGE tienen, además de parámetros obligatorios, algunos parámetros opcionales. Obtenemos la información sobre los parámetros de una función escribiendo, en una celda de cálculo, el nombre de la función seguido de un paréntesis abierto y evaluando, o, más cómodo, clicando la tecla de tabulación. Por ejemplo, tabulando tras escribir **factor**(obtenemos la información sobre la función que factoriza enteros.

Además, si sólo conocemos, o sospechamos, algunas letras del nombre de la función podemos escribirlas en la celda de cálculo y pulsar el tabulador, lo que produce un desplegable con los nombres de todas las instrucciones que comienzan por esas letras. En el desplegable podemos elegir la que nos parezca más próxima a lo que buscamos y vemos que se completan las letras que habíamos escrito con el nombre completo que hemos elegido. Si necesitamos información sobre los parámetros de la instrucción podemos añadir el interrogante al final y evaluar.

Una instrucción como **factor**($2^{137}+1$) decimos que es una función, o que está en forma funcional, ya que se parece a una función matemática que se aplica a un número entero y produce sus factores. Hay otra clase de instrucciones a las que llamamos *métodos* y que tienen una sintaxis diferente

`(objeto).metodo()`

Por ejemplo, también podemos factorizar un entero mediante

`(2^137+1).factor()`

donde $2^{137} + 1$ es un objeto de clase *número entero* al que aplicamos el método *factor*. Esta sintaxis procede de la programación *orientada a objetos* (OOP), y como Python es un lenguaje orientado a objetos es natural que en SAGE aparezca esta sintaxis.

Algunas instrucciones, como **factor**, admiten las dos formas, funciones y métodos, pero otras sólo tienen uno de las dos. Veremos ejemplos a lo largo del curso.

Como SAGE tiene miles de instrucciones preprogramadas, para toda clase de tareas en matemáticas, es muy útil tener a mano un *chuletarío* con las instrucciones más comunes¹. Hay varios y aquí puedes encontrar enlaces a algunos:

1. [En castellano.](#)
2. [Preparada por el desarrollador principal de SAGE William Stein \(en inglés\).](#)
3. [Cálculo.](#)
4. [Álgebra lineal.](#)
5. [Aritmética.](#)
6. [Estructuras algebraicas.](#)
7. [Grafos.](#)

2.1. Aritmética elemental

Las operaciones de *la escuela* son sencillas de invocar. Sobre el resultado a esperar de cada operación (o sucesión de operaciones), ha de tenerse en cuenta que la aritmética es exacta, así:

- Suma (y diferencia): $3+56+23-75$ devuelve 7.
- Producto: $3*56*23$, 3864

¹Estos *chuletaríos* pertenecen a versiones anteriores de Sage y no han sido actualizados. Aunque la sintaxis de las instrucciones de Sage no varía mucho, en ocasiones hay pequeños cambios en la versión actual respecto a la información que aparece en el chuletarío.

- Potencia: $3**2$, 9; y también lo hace 3^2 .²
- Cociente: $3864/56$ es 23. Pero, $7/3$ (o $14/6$) devolverá, $7/3$.
También la respuesta a $7/5$ se muestra en notación racional³, $7/5$, en detrimento de la notación decimal⁴, 1.4.
- División entera: $7//2$ devuelve 3, el cociente en la división entera⁵; y $7\%2$, el resto, 1.

Al encontrar varias operaciones en una misma expresión, se siguen las usuales reglas de precedencia, rotas por la presencia de paréntesis.

- $8*(5-3)+9^(1/2)+6$ da 25, mientras que $(8*(5-3)+9)^(1/2)+6$ es 11.

Estas operaciones aritméticas se efectúan en algún conjunto, generalmente un anillo o cuerpo, de números, y Sage dispone de unos cuantos predefinidos:

²La segunda expresión, con ser más sencilla, obliga a hacer aparecer el símbolo $^$ en escena, lo que, en la mayoría de teclados, requiere pulsar la barra espaciadora tras él.

³El cuerpo de los racionales, \mathbb{Q} , es suficiente para contener cocientes de números enteros.

⁴Se reserva la notación decimal para números que, con cierta precisión (finita), servirán para aproximar reales, complejos, ...

⁵*Dividendo=cociente×divisor+resto*, con $0 < \text{resto} < \text{divisor}$.

| Símbolo | Descripción |
|-----------------------------------|--|
| ZZ | anillo de los números enteros, \mathbb{Z} |
| Integers(6) | anillo de enteros módulo 6, \mathbb{Z}_6 |
| QQ | cuerpo de los números racionales, \mathbb{Q} |
| RR | cuerpo de los números reales (53 bits de precisión), \mathbb{R} |
| CC | cuerpo de los números complejos (53 bits de precisión), \mathbb{C} |
| RDF | cuerpo de números reales con doble precisión |
| CDF | cuerpo de números complejos con doble precisión |
| RealField(400) | reales con 400-bit de precisión |
| ComplexField(400) | complejos con 400-bit de precisión |
| ZZ[I] | anillo de enteros Gaussianos |
| AA, QQbar | cuerpo de números algebraicos, $\overline{\mathbb{Q}}$ |
| FiniteField(7) | cuerpo finito de 7 elementos, \mathbb{Z}_7 o \mathbb{F}_7 |
| SR | anillo de expresiones simbólicas |

2.2. Listas y tuplas

Las listas y tuplas son *estructuras de datos* y se tratarán más ampliamente en el capítulo siguiente. Para cubrir las necesidades de este capítulo baste decir que

1. Se crea una lista de nombre L escribiendo en una celda de cálculo algo como

```
L=[1,7,2,5,27,-5]
```

2. Los elementos de la lista están ordenados y se accede a ellos mediante instrucciones como $a=L[1]$, que asigna a la variable a el valor del segundo elemento de la lista, es decir a vale 7. Eso quiere decir que el primer elemento de la lista es $L[0]$ y NO $L[1]$ como podríamos esperar.
3. Las tuplas son muy parecidas a las listas, pero hay ciertas diferencias que veremos más adelante. Una tupla t se crea con la instrucción

```
t=(1,7,2,5,27,-5)
```

con paréntesis en lugar de corchetes. Se accede a sus elementos de la misma forma que para las listas, y, por ejemplo, `t[1]` vale 7. Mencionamos aquí las tuplas porque en Sage las coordenadas de un punto no forman una lista sino una tupla.

2.3. Funciones

Necesitamos definir funciones, en sentido matemático, para poder calcular con ellas, o bien representarlas gráficamente. Muchas funciones están ya definidas en Sage y lo único que haremos es asignarles un nombre cómodo para poder referirnos a ellas. Así tenemos la exponencial, el logaritmo y las funciones trigonométricas. También podemos construir nuevas funciones usando las conocidas y las operaciones aritméticas.

Podemos definir una función, por ejemplo de la variable x , mediante una expresión como `f(x)=sin(x)`, que asigna el nombre `f` a la función seno. A la derecha de la igualdad podemos escribir cualquier expresión definida usando las operaciones aritméticas y las funciones definidas en Sage. De éstas, las de uso más común son

| Símbolo | Descripción |
|------------------------------|---|
| <code>f1(x) = exp(x)</code> | que define la función exponencial, e^x |
| <code>f2(x) = log(x)</code> | que define la función logaritmo neperiano, $\ln(x)$ |
| <code>f3(x) = sin(x)</code> | que define la función seno, $\sin(x)$ |
| <code>f4(x) = cos(x)</code> | que define la función coseno, $\cos(x)$ |
| <code>f5(x) = tan(x)</code> | que define la función tangente, $\tan(x)$ |
| <code>f6(x) = sqrt(x)</code> | que define la función raíz cuadrada, \sqrt{x} |

Usando estas funciones y las operaciones aritméticas podemos definir funciones como

`F(x,y,z)=sin(x^2+y^2+z^2)+sqrt(cos(x)+cos(y)+cos(z)).`

Como definimos la función indicando explícitamente el orden de sus argumentos no existe ambigüedad sobre qué variables se deben sustituir por qué argumentos:

`f(a,b,c) = a + 2*b + 3*c`
`f(1,2,1)`

```
s = a + 2*b + 3*c
s(1,2,1)
```

```
__main__:4: DeprecationWarning: Substitution using function-call syntax
and unnamed arguments is deprecated and will be removed from a future
release of Sage; you can use named arguments instead, like EXPR(x=...,
y=...)
See http://trac.sagemath.org/5930 for details.
8
```

En este ejemplo vemos que podemos sustituir en una función, pero la sustitución en una expresión simbólica será imposible, dentro de Sage, en un futuro próximo.

Puedes encontrar más información acerca de las funciones predefinidas en Sage en esta [página](#).

La otra manera de definir funciones matemáticas es usando la misma sintaxis que utilizamos, en Python, para definir programas o trozos de programas. Esto aparecerá en detalle [más adelante](#), pero de momento podemos ver un ejemplo:

```
def f(x):
    return x*x
```

define la función *eleva al cuadrado*.

2.3.1. Variables y expresiones simbólicas

Una **variable simbólica** es un objeto en Python que representa una *variable*, en sentido matemático, que puede tomar cualquier valor en un cierto dominio. Casi siempre, ese dominio es un cuerpo de números, como los racionales, los reales o los números complejos. En el caso de los números racionales la representación es exacta, mientras que los reales, o los complejos, se representan usando decimales con un número dado de dígitos en la parte decimal.

1. La variable simbólica x está predefinida, y para cualquier otra que utilicemos debemos avisar al intérprete:

```
var('a b c')
```

- La igualdad $a = 2$ es una asignación que crea una variable a y le da el valor 2. Hasta que no se evalúe una celda que le asigne otro valor, por ejemplo $a = 3$, el valor de a será 2. Los valores de las variables, una vez asignados, se mantienen dentro de la hoja mientras no se cambien explícitamente.
- Una operación que involucra una o más variables simbólicas no devuelve un valor numérico, sino una **expresión simbólica** que involucra números, operaciones, funciones y variables simbólicas.

```
s = a + b + c
s2 = a^2 + b^2 + c^2
s3 = a^3 + b^3 + c^3
s4 = a + b + c + 2*(a + b + c)
p = a*b*c
```

- Una igualdad como $s = a + b + c$ es, de hecho, una asignación: después de ejecutar esa línea el valor de s es $a + b + c$ y, por tanto, el de s^2 es $(a + b + c)^2$, etc.
- Podemos imprimirlas como código

```
print s; s2; s3; s4; p
```

```
a + b + c
a^2 + b^2 + c^2
a^3 + b^3 + c^3
3*a + 3*b + 3*c
a*b*c
```

o mostrarlas en un formato matemático más habitual

```
show([s, s2, s3, s4, p])
```

```
 $[a + b + c, a^2 + b^2 + c^2, a^3 + b^3 + c^3, 3a + 3b + 3c, abc]$ 
```

- Si en algún momento sustituimos las variables simbólicas por números (o elementos de un anillo), podremos realizar las operaciones y obtener un número (o un elemento de un anillo).

```
print s(a=1, b=1, c=1), s(a=1, b=2, c=3)
s(a=1, b=1, c=1)+ s(a=1, b=2, c=3)
```

```
3 6
9
```

La sustitución no cambia el valor de s , que sigue siendo una expresión simbólica, y sólo nos da el valor que se obtiene al sustituir. Obsérvese, en cambio, qué se obtiene si ejecutamos

```
a=1;b=1;c=1
print s
```

7. Si operamos expresiones simbólicas, obtenemos otras expresiones simbólicas, aunque pocas veces estarán simplificadas

```
ex = (1/6)*(s^3 - 3*s*s2 + 2*s3 )
show(ex)
```

$$\frac{1}{6} (a + b + c)^3 + \frac{1}{3} a^3 + \frac{1}{3} b^3 + \frac{1}{3} c^3 - \frac{1}{2} (a + b + c)(a^2 + b^2 + c^2)$$

2.3.1.1. Simplificar expresiones

Observamos que al crear la expresión se han realizado “*de oficio*” algunas simplificaciones triviales. En ejemplos como el de arriba, nos puede interesar simplificar la expresión todavía más, pero es necesario decir qué queremos exactamente.

Existen varias estrategias para intentar simplificar una expresión, y cada estrategia puede tener más o menos éxito dependiendo del tipo de expresión simbólica. Algunas dan lugar a una expresión más sencilla en algunos casos, pero no en otros, y con expresiones complicadas pueden consumir bastante tiempo de proceso. Para la expresión anterior, como tenemos un polinomio, es buena idea expandirla en monomios que se puedan comparar unos con otros, usando el método `.expand()`.

```
show(ex.expand())
```

```
abc
```


A menudo nos interesa lo contrario: factorizar la expresión usando `.factor()`

```
p = a^3 + a^2*b + a^2*c + a*b^2 + a*c^2 + b^3 + b^2*c + b*c^2 + c^3
show(p)
show(p.factor())
```

$$a^3 + a^2b + a^2c + ab^2 + ac^2 + b^3 + b^2c + bc^2 + c^3$$

$$(a + b + c)(a^2 + b^2 + c^2)$$

Si consultas con el tabulador los métodos de las expresiones simbólicas, verás que hay métodos específicos para expresiones con funciones trigonométricas, exponenciales, con radicales o fracciones (es decir, con funciones racionales), ...

`p.simplify`

evaluate

`p.simplify`

`p.simplify_exp`

`p.simplify_factorial`

`p.simplify_full`

`p.simplify_log`

`p.simplify_radical`

`p.simplify_rational`

`p.simplify_trig`

```
p = sin(3*a)
show(p.expand_trig())
```

$$-\sin(a)^3 + 3 \sin(a) \cos(a)^2$$

```
p = sin(a)^2 - cos(a)^2
show(p.simplify_trig())
```

$$-2 \cos(a)^2 + 1$$

```
p = 2^a * 4^(2*a)
show(p.simplify_exp())
```

$$2^{5a}$$

```
p = 1/a - 1/(a+1)
show(p.simplify_rational())
```

$$\frac{1}{a^2 + a}$$

2.3.2. Variables booleanas

Un tipo especial de variables es el *booleano* o lógico. Una tal variable toma valores **True** (verdadero) o **False** (falso).⁶

El doble signo igual (==) sirve para comparar, y devuelve **True** o **False** según los objetos comparados sean iguales o no para el intérprete. De la misma manera se pueden usar, siempre que tengan sentido, las comparaciones $a < b$ y $a \leq b$. Como veremos en el capítulo 4, estas comparaciones aparecen en los bucles **while** y al bifurcar la ejecución de código mediante un **if**. Operaciones básicas con variables booleanas son la *conjunción* (**and**), la *disyunción* (**or**) y la *negación* (**not**):

| and | True | False |
|------------|-------|-------|
| True | True | False |
| False | False | False |

| or | True | False |
|-----------|------|-------|
| True | True | True |
| False | True | False |

| | not |
|-------|------------|
| True | False |
| False | True |

2.4. Gráficas

2.4.1. Gráficas en 2D

Utilizaremos comandos como los que siguen para obtener objetos gráficos:

1. **point**(punto o lista), **points**(lista), **point2d**(lista), **point3d**(lista) dibujan los puntos de la lista que se pasa como argumento. Usaremos este tipo de gráficas, en particular, al estudiar la paradoja de Bertrand en el capítulo 11.

⁶Aunque, como veremos algo más adelante, también los valores 1, en lugar de **True**, y 0 en lugar de **False**.

2. **line**(lista), **line2d**(lista), **line3d**(lista) dibujan líneas entre los puntos de la lista que se pasa como argumento. Usaremos este tipo de gráficas, en particular, al estudiar el problema conocido como la ruina del jugador en el capítulo 11.
3. **plot**(f,x,xmin=a,xmax=b) esboza una gráfica de la función de una variable f en el intervalo $[a, b]$. La escala en el eje OY la elige Sage automáticamente. En ocasiones, debido a esa elección automática, la gráfica mostrada consiste en un trozo encima del eje OX y otro prácticamente vertical, lo que no resulta muy útil. Podemos centrarnos en la parte que nos interesa del plano mediante **plot**(f,x,xmin=a,xmax=b,ymin=c,ymax=d), que muestra la parte de la gráfica contenida en el rectángulo $[a, b] \times [c, d]$.
4. **parametric_plot**([f(t),g(t)],(t,0,2)) esboza una gráfica de la curva dada en paramétricas mediante dos funciones de t , f y g , con la variable t variando en el intervalo $[0, 2]$.
5. Mediante **implicit_plot**(f,(x,-5,5),(y,-5,5)) obtenemos una representación de la parte de la curva $f(x, y) = 0$ contenida en el cuadrado $[-5, 5] \times [-5, 5]$. Decimos que se trata de una representación de una curva “*en implícitas*”.

Puedes ver algunos ejemplos de gráficas en el plano en la hoja de Sage [21-CAVAN-graficas.ipynb](#).

2.4.2. Gráficas en 3D

El sistema para representaciones gráficas en 3D que tiene Sage no funciona en **jupyterhub**, es decir no muestra los gráficos cuando se ejecuta en el servidor del Departamento. En cambio, desde la versión **8.0** de Sage, funciona bien cuando se ejecuta en la versión local de **Jupyter**. Entonces, y hasta que se resuelva este problema, en el servidor usaremos **matplotlib**.

1. En general, **matplotlib** produce gráficos 3D a partir de listas de coordenadas de puntos, de la forma $(x, y, f(x, y))$, en \mathbb{R}^3 . Así por ejemplo, para representar la gráfica de una función de dos variables $f(x, y)$ primero necesitamos una lista con coordenadas de puntos en el rectángulo $[a, b] \times [c, d]$ sobre el que deseamos obtener la gráfica.

```
x = np.arange(-2,2,0.05)
y = np.arange(-2,2,0.05)
X,Y = np.meshgrid(x, y)
```

Esto nos dará $80 = 4/0.05$ puntos en el intervalo $[-2, 2]$ para la variable x , otros 80 para la y , y un retículo de 6400 puntos, uniformemente distribuidos, en el cuadrado $[-2, 2] \times [2, 2]$. Los comandos que empiezan por **np** son de **numpy**, y permiten la manipulación eficiente de listas y matrices. Veremos más sobre esto más adelante.

2. Ahora definimos la función que vamos a representar

```
def f(x,y):
    return np.cos(x**2+y**2)
```

Como usamos listas y matrices de **numpy**, las funciones matemáticas como el coseno también deben ser de **numpy**, y por eso usamos `np.cos`.

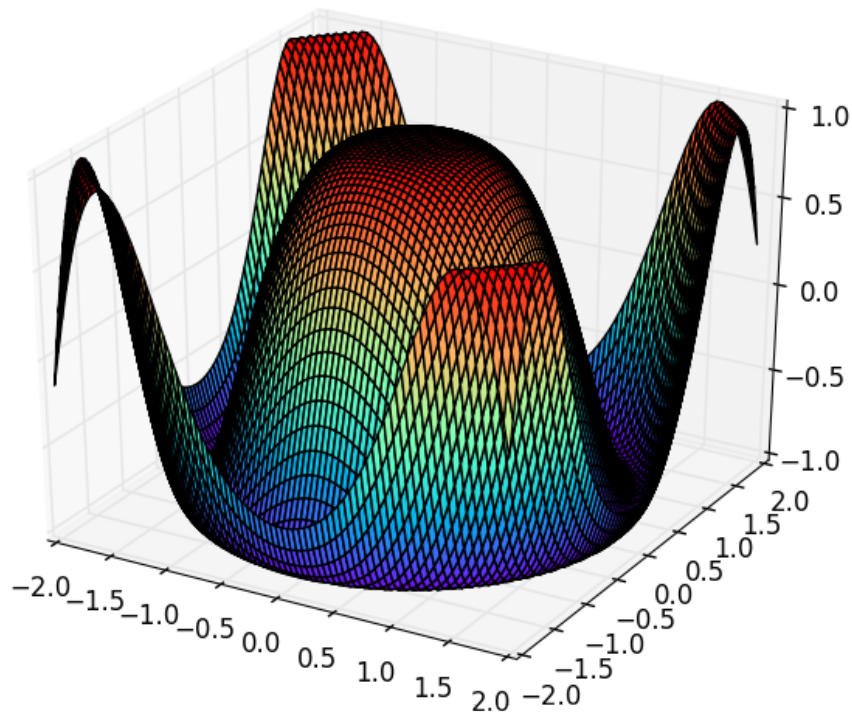
3. Finalmente, representamos la superficie mediante, por ejemplo,

```
1 fig = plt.figure()
2 ax = fig.add_subplot(111, projection='3d')
3 zs = np.array([f(x,y) for x,y in zip(np.ravel(X), np.ravel(Y))])
4 Z = zs.reshape(X.shape)
5 ax.plot_surface(X,Y,Z, cmap=plt.cm.rainbow,rstride=1, cstride=1)
```

La tercera línea es la que crea las coordenadas de los puntos en \mathbb{R}^3 que usamos para construir la superficie, y la quinta es la que finalmente genera el gráfico interactivo.

4. Para obtener el gráfico primero, es decir en la primera celda que ejecutamos, hay que importar una serie de paquetes

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from math import *
%matplotlib notebook
```



Puedes ver un resumen del uso de `matplotlib` en [este tutorial](#)⁷ y algunos ejemplos en la hoja

[21-CAVAN-graficas-matplotlib.ipynb](#).

2.4.3. Gráficas 3D en el notebook antiguo

Las tres funciones principales para representaciones gráficas 3D son

1. `plot3d(g(x,-10,10),(y,-10,10))` esboza una gráfica de la función de dos variables g sobre el cuadrado $[-10, 10] \times [-10, 10]$.
2. `parametric_plot3d([f(u,v),g(u,v),h(u,v)],(u,0,2),(v,0,2))` esboza una gráfica de la superficie dada en paramétricas mediante tres funciones de u y v , f , g y h , con las variables u y v en el intervalo $[0, 2]$.

Los puntos de la superficie son entonces los que se obtienen mediante $x = f(u, v)$, $y = g(u, v)$, $z = h(u, v)$.

3. Con `implicit_plot3d(f(x,-5,5),(y,-5,5),(z,-5,5))` se representa una parte de la superficie $f(x, y, z) = 0$ contenida en el cubo $[-5, 5] \times [-5, 5] \times [-5, 5]$. Decimos que se trata de una representación de una superficie “*en implícitas*”.

Si se ejecutan en **Jupyter** se obtiene un gráfico vacío, pero en el **notebook** antiguo de Sage (1.1) funcionan perfectamente.

2.5. Cálculo

En esta sección usaremos la hoja de Sage [22-CAVAN-calculo-s1.ipynb](#).

Cuando usamos el ordenador para estudiar problemas de cálculo diferencial o integral operamos frecuentemente con valores aproximados de los números reales implicados, es decir truncamos los números reales después de un número prefijado de decimales.

Sin embargo, en Sage podemos realizar gran cantidad de operaciones de manera “simbólica”, es decir, sin evaluar de manera aproximada los números reales o complejos implicados. Así, por ejemplo, `sqrt(2)` es una representación simbólica de $\sqrt{2}$, una expresión cuyo cuadrado es exactamente 2, mientras que `sqrt(2).n()` (o bien `n(sqrt(2))`) es un valor aproximado con 20

⁷IPython es el nombre antiguo de **Jupyter**, pero la mayor parte de lo que se dice en este tutorial funciona en **Jupyter** sin problema.

decimales y su cuadrado no es exactamente 2. Para obtener una representación decimal de una expresión simbólica, podemos usar los comandos `n()`, `N()`, o los métodos homónimos `.n()`, `.N()` (`n` de numérico).⁸

El cálculo con valores aproximados de números reales necesariamente introduce errores, y suele llamarse *Cálculo numérico* a la materia que estudia cómo controlar esos errores, de manera que conocemos el grado de validez del resultado final y tratamos de que sea lo más alto posible. Volveremos en el capítulo 7 sobre este asunto.

La diferencia entre el *cálculo simbólico* y el *cálculo numérico* es importante: el cálculo simbólico es exacto pero más limitado que el numérico. Un ejemplo típico podría ser el cálculo de una integral definida

$$\int_a^b f(x) \, dx, \quad (2.1)$$

que simbólicamente requiere el cálculo de una primitiva $F(x)$ de $f(x)$ de forma que el valor de la integral definida es $F(b) - F(a)$. El cálculo de primitivas no es fácil, y para algunas funciones, como por ejemplo $\sin(x)/x$, la primitiva no se puede expresar usando las funciones habituales y, si fuera necesario, deberíamos considerarla como una nueva función elemental semejante a las trigonométricas o la exponencial.

En cambio, para cada función continua $f(x)$ en un intervalo $[a, b]$ podemos calcular fácilmente un valor aproximado para la integral definida (2.1). Veremos alguno de estos métodos en la sección 11.5.

2.5.1. Ecuaciones

1. El comando `solve()` permite resolver ecuaciones (al menos lo intenta): para resolver una ecuación llamamos a `solve()` con la ecuación como primer argumento y la variable a despejar como segundo argumento, y recibimos una lista con las soluciones.

#Las soluciones de esta ecuación cúbica son números complejos
`solve(x^3 - 3*x + 5, x)`

2. El algoritmo resuelve también un sistema de ecuaciones. Basta pasar la *lista* de igualdades o expresiones.

⁸Es muy habitual la asignación ' $n = '$, o ' $N = '$, cuando pensamos en dar nombre a una variable entera. Si la utilizamos en una hoja de Sage, el sistema no nos avisa de que estamos *tapando las funciones* `n()`, `N()`. Desde ese momento, y hasta que no reiniciemos la hoja, ya no funcionará la función. Se recomienda el uso de los métodos, `.n()`, `.N()`, para evitar innecesarios dolores de cabeza.

```
var('x y')
solve([x^2+y^2==1,x-y+1],x,y)
```

3. El comando `solve()` intenta obtener soluciones exactas en forma de expresión simbólica, de la ecuación o sistema, y frecuentemente no encuentra ninguna. También es posible buscar soluciones aproximadas mediante métodos numéricos: el comando `find_root(f,a,b)` busca una solución de la ecuación $f(x) = 0$ perteneciente al intervalo $[a, b]$. Volveremos sobre este asunto en la sección 7.6.

2.5.2. Límites

1. El método `.limit()` (o la función `limit()`) permite calcular límites de funciones. Para calcular el límite de `f` en un punto:

```
f1=x^2
print f1.limit(x=1)
```

2. También se puede calcular el límite cuando la variable tiende a infinito

```
f2=(x+1)*sin(x)
f3=f2/f1
f3.limit(x=+oo)
```

3. Si Sage sabe que el límite no existe, por ejemplo para la función $f(x) = 1/(x * \sin(x))$, la función `limit()` devuelve el valor `und`, abreviatura de *undefined*, o el valor `ind`, que indica que no existe límite pero la función permanece acotada, por ejemplo $\sin(x)$ cuando x tiende a infinito, cerca del valor límite de la variable.

Por último, en algunos casos Sage no sabe cómo determinar el límite o no sabe que no existe, y entonces devuelve la definición de la función. Esto ocurre, por ejemplo, con la función $f(x) := x/\sin(1/x)$ cuando se quiere calcular el límite cuando x tiende a cero.

4. También podemos calcular así límites de sucesiones. Al fin y al cabo, si la expresión simbólica admite valores reales, el límite de la función que define en infinito, si existe, es el mismo que el límite de la sucesión de naturales definido por la misma expresión.

2.5.3. Series

Una serie es un tipo particular de sucesión: dada una sucesión a_0, a_1, a_2, \dots queremos dar sentido a la suma infinita

$$S = a_0 + a_1 + a_2 + \dots + a_n + \dots$$

Definimos S como el límite, si existe, de la sucesión de “sumas parciales” $S_n := a_0 + a_1 + a_2 + \dots + a_n$ cuando n tiende a infinito y decimos que S es la suma de la serie.

Para que pueda existir un límite para S_n tiene que ocurrir que el límite de a_n , cuando n tiende a infinito, sea cero. Es una condición necesaria pero NO SUFICIENTE para la existencia de suma de la serie. Por ejemplo, la sucesión $a_n := 1/n$ tiende a cero cuando n tiende a infinito, pero el límite de la correspondiente sucesión de sumas parciales es infinito.

El comando `sum()`, tomando ∞ como límite superior, permite, a veces, calcular la suma de una serie infinita:

`sum(expresion, variable, limite_inferior, limite_superior)`

```
sum(1/k^2, k, 1, oo).show()
```

$$\frac{1}{6}\pi^2$$

En casos como el anterior, Sage no realiza ningún cálculo para devolvernos su respuesta. Lo único que hace es identificar la serie que queremos sumar y nos devuelve el valor de la suma si lo conoce. Por ejemplo, si le pedimos la suma de los inversos de los cubos de enteros nos dice que vale $\text{zeta}(3)$, que no es sino el nombre que se usa en Matemáticas para esa suma. Estudiaremos en detalle el cálculo con series en el capítulo 7.

2.5.4. Cálculo diferencial

1. El método `.derivative()`, o la función `derivative()`, permite calcular derivadas de funciones simbólicas. Las **derivadas** se obtienen siguiendo metódicamente las reglas de derivación y no suponen ningún problema al ordenador:

```
f=1/(x*sin(x))
g=f.derivative()
show([g,g.simplify_trig()])
```

2. Se pueden calcular derivadas de órdenes superiores

```
m=5
[derivative(x^m,x,j) for j in range(m+1)]
```

Para derivar funciones de varias variables, es decir calcular derivadas parciales (i.e. derivar una función de varias variables respecto a una de ellas manteniendo las otras variables en valores constantes), basta especificar la variable con respecto a la que derivamos y el orden hasta el que derivamos:

```
F(x,y)=x^2*sin(y)+y^2*cos(x)
show([F.derivative(y,2),derivative(F,x,1).derivative(y,1)])
```

2.5.5. Desarrollo de Taylor

Así como la derivada de una función en un punto x_0 nos permite escribir una aproximación lineal de la función CERCA DEL PUNTO

$$f(x) \sim f(x_0) + f'(x_0)(x - x_0),$$

el polinomio de Taylor nos da una aproximación polinomial, de un cierto grado prefijado, válida CERCA DEL PUNTO. Así por ejemplo:

```
f(x)=exp(x)
taylor(f,x,0,20)
```

nos devuelve un polinomio de grado 20 que cerca del origen aproxima la función exponencial. Volveremos sobre este asunto en el capítulo 7.

2.5.6. Cálculo integral

El cálculo de *primitivas* es mucho más complicado que la derivación, ya que no existe ningún método que pueda calcular la primitiva de cualquier función. En realidad, hay muchas funciones elementales (construidas a partir de funciones trigonométricas, exponenciales y algebraicas mediante sumas, productos y composición de funciones) cuyas primitivas, aunque estén bien definidas, no se pueden expresar en términos de estas mismas funciones. Los ejemplos $f(x) = e^{-x^2}$ y $f(x) = \frac{\sin(x)}{x}$ son bien conocidos.

Aunque en teoría existe un algoritmo (el algoritmo de [Risch](#)) capaz de decidir si la primitiva de una función elemental es elemental, dificultades prácticas imposibilitan llevarlo a la práctica, y el resultado es que incluso en casos en los que integramos una función cuya primitiva es una función elemental, nuestro algoritmo de integración simbólica puede no darse cuenta. Si Sage no puede calcular una primitiva de $f(x)$ explícita devuelve

$$\int f(x) dx.$$

1. Los métodos (funciones) para el cálculo de primitivas a nuestra disposición son: `.integral()` e `.integrate()`.

```
f=1/sin(x)
show(f.integrate(x))
```

2. Siempre podemos obtener una aproximación numérica de una integral definida:

```
f=tan(x)/x
[numerical_integral(f,pi/5,pi/4),N(f.integrate(x,pi/5,pi/4))]
```

Obsérvese que, con `numerical_integral()`, el intérprete devuelve una tupla, con el valor aproximado de la integral y una cota para el error.

3. También es posible [integrar numéricamente funciones de varias variables](#), aunque, de momento, no vamos a tratar el tema. Así como las integrales de funciones de una variable corresponden al cálculo de áreas, las integrales de funciones de varias variables permiten calcular volúmenes e hipervolumenes.

En el capítulo 11 veremos un método, conocido como integración de Monte Carlo que permite calcular valores aproximados de integrales de funciones de una o varias variables.

2.6. Álgebra lineal

Podemos decir que el Álgebra Lineal, al menos en espacios de dimensión finita, trata de la *resolución de todos aquellos problemas que pueden reducirse a encontrar las soluciones de un sistema de ecuaciones de primer grado en todas las incógnitas*,

es decir, un sistema lineal de ecuaciones. Es posible realizar todas las operaciones necesarias para resolver tales sistemas mediante cálculo con matrices, y esa es la forma preferida para resolver problemas de Álgebra Lineal mediante ordenador.

Los sistemas de ecuaciones lineales con coeficientes en un cuerpo, por ejemplo el de los números racionales, siempre se pueden resolver y encontrar explícitamente todas las soluciones del sistema en ese cuerpo o en uno que lo contenga. Esto no es cierto para ecuaciones polinomiales de grado más alto, y, por ejemplo, no hay métodos generales para resolver una única ecuación polinomial, de grado arbitrario n , en una única variable

$$a_0 + a_1x + a_2x^2 + \cdots + a_nx^n = 0.$$

Veremos en la parte final del curso algo de los que se puede decir acerca de la resolución de los sistemas de ecuaciones polinomiales.

2.6.1. Construcción de matrices

El constructor básico de matrices en Sage es la función `matrix()`, que, en su forma más simple, tiene como argumentos

1. El conjunto de números, enteros, racionales, etc., al que pertenecen las entradas de la matriz.
2. Una lista cuyos elementos son listas de la misma longitud, y que serán las *filas* de la matriz.

```
A=matrix(ZZ,[[1,2,3],[4,5,6]]);A;show(A)
```

Una variante útil contruye la misma matriz con

```
A=matrix(ZZ,2,[1,2,3,4,5,6]);show(A)
```

que trocea la lista dada como tercer argumento en el número de filas dado por el segundo, en este caso 2 filas. Por supuesto, el número de elementos de la lista debe ser múltiplo del segundo argumento o aparece un error.

Para más ejemplos e información sobre `matrix()`, pulsar el **tabulador** del teclado tras escribir `matrix(` en una celda.

2.6.2. Submatrices

Sobre el acceso a las entradas de una matriz se ha de tener en cuenta que tanto filas como columnas empiezan su numeración en 0. Así, en la matriz 3×2 definida por $A=\text{matrix}(3,\text{range}(6))$, es decir

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{pmatrix}$$

el extremo superior izquierdo tiene *índices* ‘0,0’, y el inferior derecho, ‘2,1’. Una vez claros los índices de un elemento, Sage usa la notación *slice* (por cortes) de Python para acceder a él: $A[0][0]$ y $A[2][1]$ serían los elementos recién mencionados. Para el caso especial de matrices, se ha adaptado también una notación más habitual: $A[0,0]$ y $A[2,1]$.

En la notación *slice*, dos índices separados por dos puntos, $i:j$, indican el corte entre el primer índice, incluido, hasta el segundo, que se excluye. En el ejemplo, se muestran las filas de índices 2 y 3 de la matriz. Si se excluye alguno de los índices, se toma, por defecto, el valor más extremo.

```
A=matrix(ZZ,10,[1..100])
```

```
A[7:]
```

```
[ 71 72 73 74 75 76 77 78 79 80]
[ 81 82 83 84 85 86 87 88 89 90]
[ 91 92 93 94 95 96 97 98 99 100]
```

La matriz $A[7:]$ consiste en las filas desde la octava, que tiene índice 7 porque hemos empezado a contar en 0, hasta la décima.

```
A=matrix(10,[1..100])
```

```
A[:2]
```

```
[ 1 2 3 4 5 6 7 8 9 10]
[11 12 13 14 15 16 17 18 19 20]
```

La matriz $A[:2]$ consiste en las filas primera y segunda, es decir, las de índice menor estrictamente que 2. Utilizando el doble índice, podemos recortar recuadros más concretos:

```
A=matrix(10,[1..100])
```

```
A[3:5,4:7]
```

$$\begin{bmatrix} 35 & 36 & 37 \\ 45 & 46 & 47 \end{bmatrix}$$

Unas últimas virguerías gracias a la notación slice: saltos e índices negativos. La matriz A es, como en el ejemplo anterior, la matriz 10×10 con los primeros cien enteros colocados en orden en las filas de A .

```
A[2:5:2,4:7:2]
```

produce

$$\begin{bmatrix} 25 & 27 \\ 45 & 47 \end{bmatrix}$$

se queda con las filas con índice del 2 al 4, saltando de dos en dos debido al 2 que aparece en tercer lugar en 2:5:2, y con las columnas con índices del 4 al 6 también saltando de dos en dos por el que aparece en 4:7:2. Siempre hay que recordar que las filas y columnas se numeran empezando en el cero.

Por otra parte, también podemos usar índices negativos lo que conduce a quedarnos con filas o columnas que contamos hacia atrás, de derecha a izquierda, empezando por las últimas:

```
A[-1];A.column(-2)
```

$$\begin{aligned} &(91, 92, 93, 94, 95, 96, 97, 98, 99, 100) \\ &(9, 19, 29, 39, 49, 59, 69, 79, 89, 99) \end{aligned}$$

La primera fila del resultado, que corresponde a $A[-1]$, es la última fila de la matriz, y la segunda es la novena columna (que tiene índice 8). La última columna se obtendría con $A.\text{column}(-1)$.

2.6.3. Operaciones con matrices

Las operaciones entre matrices, suma, diferencia, multiplicación, potencia e inversa, se denotan con los mismos símbolos que las correspondientes operaciones entre números. En el caso de operaciones entre matrices es posible que la operación no sea posible porque los tamaños de las matrices implicadas no sean compatibles, o en el caso de la inversa porque la matriz no sea cuadrada de rango máximo.

De entre todas las funciones y métodos que se aplican a matrices hemos seleccionado las que parecen más útiles:

- `identity_matrix(n)`.- matriz identidad $n \times n$

- `A.det()`, `det(A)`.- determinante de A
- `A.rank()`, `rank(A)`.- rango de A
- `A.trace()`.- traza de A
- `A.inverse()`.- inversa de A
- `A.transpose()`.- traspuesta de A
- `A.adjoint()`.- matriz adjunta de A
- `A.echelonize()`, `A.echelon_form()`.- matriz escalonada de A , en el menor anillo en que vivan sus entradas
- `A.rref()`.- matriz escalonada de A , en el menor cuerpo en que coincidan sus entradas

Dado que resolver un sistema lineal de ecuaciones consiste, desde un punto de vista matricial, en obtener la forma escalonada de la matriz (reducción gaussiana), la instrucción quizá más importante en la lista es `A.echelon_form()`.

Aunque `A.echelonize()` y `A.echelon_form()` hacen esencialmente lo mismo, la primera deja la forma escalonada en la variable A , por tanto machaca el valor antiguo de A , y no es posible asignar el resultado a otra variable, es decir $B = A.echelonize()$ no funciona, mientras que la segunda forma funciona como esperamos $B=A.echelon_form()$ hace que la forma escalonada quede en B y la matriz A todavía existe con su valor original. Puedes comprobar este comportamiento en la hoja [23-CAVAN-reduccion-gaussiana.ipynb](#).

2.6.4. Espacios vectoriales

Operar con matrices en el ordenador *siempre* ha sido posible debido a que los lenguajes de programación disponen de la estructura de datos `array`, que esencialmente es lo mismo que una matriz. En los sistemas de cálculo algebraico, como Sage, existe la posibilidad de definir objetos matemáticos mucho más abstractos como, por ejemplo, [grupos](#), [espacios vectoriales](#), [grafos](#), [anillos](#), etc.

Nos fijamos en esta subsección en el caso de los espacios vectoriales. ¿Qué significa definir, como objeto de Sage, un espacio vectorial E de dimensión 3 sobre el cuerpo de los números racionales?

Lo definimos mediante $E = \text{VectorSpace}(\mathbb{Q}, 3)$ y sus elementos, vectores, mediante, por ejemplo, $v = \text{vector}([1, 2, 3])$ o bien $v = E([1, 2, 3])$.

Una vez definido el espacio vectorial y vectores en él, podemos realizar cálculos con vectores siguiendo las reglas que definen, en matemáticas, los espacios vectoriales. Así por ejemplo, el sistema sabe que $\mathbf{v} + (-1) * \mathbf{v} = \mathbf{0}$.

También es posible construir espacios de matrices con una instrucción como $M = \text{MatrixSpace}(\mathbb{Q}\mathbb{Q}, 4, 4)$, que define M como el conjunto de matrices con coeficientes racionales 4×4 , con operaciones de suma, producto por escalares, producto de matrices y matriz inversa.

2.6.5. Subespacios vectoriales

1. Podemos definir fácilmente el subespacio engendrado por un conjunto de vectores, y después realizar operaciones como intersección o suma de subespacios, o comprobaciones como igualdad o inclusión de subespacios.

```
V1 = VectorSpace(QQ,3)
v1 = V1([1,1,1])
v2 = V1([1,1,0])
L1 = V1.subspace([v1,v2])
print L1
```

2. Definido un subespacio, podemos averiguar información sobre él:

```
print dim(L1); L1.degree(); L1.ambient_vector_space()
```

Sage llama *degree* de un subespacio a la dimensión de su espacio ambiente.

3. Muchos operadores actúan sobre subespacios vectoriales, con los significados habituales.

```
# Pertenencia a un subespacio
v3, v4 = vector([1,0,1]), vector([4,4,3])
print v3 in L1; v4 in L1
```

```
#Comprobación de igualdad
print L1 == V1
print L1 == V1.subspace([v1,v1+v2])
```



```
#Comprobación de inclusión
```

```
print L1 <= V1; L1 >= V1; L1 >= V1.subspace([v1])
```

```
#Intersección y suma de subespacios
```

```
L1 = V1.subspace([(1,1,0),(0,0,1)])
```

```
L2 = V1.subspace([(1,0,1),(0,1,0)])
```

```
L3 = L1.intersection(L2)
```

```
print '* Intersección de subespacios: '
```

```
print L3
```

```
L4 = L1+L2
```

```
print '* Suma de subespacios: ';L4
```

2.6.6. Bases y coordenadas

Como hemos visto, se define un subespacio de un espacio vectorial en Sage mediante un conjunto de generadores del subespacio, pero internamente se guarda mediante una base del subespacio. Esta base, en principio no es un subconjunto del conjunto generador utilizado para definir el subespacio, pero podemos imponer que lo sea con el método **subspace_with_basis**:

```
L1 = V1.subspace_with_basis([v1,v2,v3])
```

```
print L1.basis(); L1.basis_matrix()
```

El método **.coordinates()** nos da las coordenadas de un vector en la base del espacio

```
print (L1.coordinates(v1), L2.coordinates(v1))
```

2.6.7. Producto escalar

Podemos definir un espacio vectorial con una forma bilineal mediante

```
V = VectorSpace(QQ,2, inner_product_matrix=[[1,2],[2,1]])
```

Acabamos con una lista, incompleta, de funciones y métodos relacionados con el producto escalar de vectores.

- u.**dot_product**(v).- producto escalar, $u \cdot v$
- u.**cross_product**(v).- producto vectorial, $u \times v$
- u.**pairwise_product**(v).- producto elemento a elemento
- **norm**(u), u.**norm**(), u.**norm**(2).- norma Euclídea
- u.**norm**(1).- suma de coordenadas en valor absoluto
- u.**norm**(Infinity).- coordenada con mayor valor absoluto
- u.**inner_product**(v).- producto escalar utilizando la matriz del producto escalar

2.7. Ejercicios

2.7.1. Inducción y sucesiones

Ejercicio 1. Demuestra por inducción sobre $n \in \mathbb{N}$ las afirmaciones siguientes:

1. $1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}.$
2. $\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \cdots + \frac{1}{n \cdot (n+1)} = \frac{n}{n+1},$ si $n \geq 1.$
3. $1 \cdot 1! + 2 \cdot 2! + \cdots + n \cdot n! = (n+1)! - 1.$
4. $\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + \frac{n}{2^n} = 2 - \frac{n+2}{2^n}.$
5. $(1+q)(1+q^2)(1+q^4) \cdots (1+q^{2^n}) = \frac{1-q^{2^{n+1}}}{1-q}.$

En este ejercicio se pide una “demostración matemática” por inducción. Más adelante **veremos** cómo programar la comprobación, hasta un n prefijado, de fórmulas copmo estas.

Ejercicio 2. La sucesión de Fibonacci, $\{F_n\}$, está definida por medio de la ley de recurrencia:

$$F_1 = 1, \quad F_2 = 1, \quad F_{n+2} = F_n + F_{n+1}.$$

Calcular los diez primeros términos de la sucesión y comprobar, para $1 \leq n \leq 10$, la siguiente identidad:

$$F_n = \frac{\left[\frac{1+\sqrt{5}}{2}\right]^n - \left[\frac{1-\sqrt{5}}{2}\right]^n}{\sqrt{5}}.$$

La sucesión de Fibonacci vuelve a aparecer varias veces a lo largo del curso, sobre todo en la página **138** y siguientes.

Ejercicio 3. Demostrar por inducción la fórmula para la suma de los primeros n cubos

$$1^3 + 2^3 + \dots + n^3 = \frac{(n+1)^2 n^2}{4}.$$

En el ejercicio **13** se plantea un método para encontrar esta fórmula.

Ejercicio 4. Estudiar el límite de las siguientes sucesiones

$$\begin{aligned} & \text{(a) } \left\{ \frac{n^2}{n+2} \right\}; & \text{(b) } \left\{ \frac{n^3}{n^3+2n+1} \right\}; & \text{(c) } \left\{ \frac{n}{n^2-n-4} \right\}; & \text{(d) } \left\{ \frac{\sqrt{2n^2-1}}{n+2} \right\}; \\ & \text{(e) } \left\{ \frac{\sqrt{n^3+2n+n}}{n^2+2} \right\}; & \text{(f) } \left\{ \frac{\sqrt{n+1}+n^2}{\sqrt{n+2}} \right\}; & \text{(g) } \left\{ \frac{(-1)^n n^2}{n^2+2} \right\}; & \text{(h) } \left\{ \frac{n+(-1)^n}{n} \right\}; \\ & \text{(i) } \left\{ \left(\frac{2}{3}\right)^n \right\}; & \text{(j) } \left\{ \left(\frac{5}{3}\right)^n \right\}; & \text{(k) } \left\{ \frac{2^n}{4^{n+1}} \right\}; & \text{(l) } \left\{ \frac{3^n+(-2)^n}{3^{n+1}+(-2)^{n+1}} \right\}; \\ & \text{(m) } \left\{ \frac{n}{n+1} - \frac{n+1}{n} \right\}; & \text{(n) } \left\{ \sqrt{n+1} - \sqrt{n} \right\}; & \text{(ñ) } \left\{ \frac{1}{n^2} + \frac{2}{n^2} + \dots + \frac{n}{n^2} \right\}. \end{aligned}$$

Atención: la sucesión $\frac{3^n+(-2)^n}{3^{n+1}+(-2)^{n+1}}$ tiene límite $\frac{1}{3}$. El siguiente código

```
var('m')
l(m)=3^m+(-2)^m
ll=l(m)/l(m+1)
ll.limit(m=oo)
```

devuelve, incorrectamente, 0, pero es fácil ayudar al intérprete:

```
var('m')
l(m)=3^m+(-2.0)^m
ll=l(m)/l(m+1)
ll.limit(m=oo)
```

1/3

¿Cuál puede ser la explicación?

Ejercicio 5. Calcular, si existen, los límites de las sucesiones que tienen como término general

$$a_n = \left(\frac{n^2 + 1}{n^2} \right)^{2n^2 - 3}, \quad b_n = \left(\frac{n^2 - 1}{n^2} \right)^{2n^2 + 3}, \quad c_n = a_n + \frac{1}{b_n}.$$

2.7.2. Un ejercicio de cálculo

Consideramos el polinomio en la variable x y dependiente de dos parámetros reales, a y b , dado por

$$p(x, a, b) := x^4 - 6x^2 + ax + b,$$

y queremos estudiar el número de raíces reales que tiene dependiendo del valor de los parámetros.

1. Es un teorema importante, llamado en ocasiones el *teorema fundamental del álgebra*, el hecho cierto de que todo polinomio de grado n con coeficientes complejos tiene exactamente n raíces complejas si se cuentan con sus multiplicidades.
2. Si un polinomio tiene coeficientes reales entonces tiene un número par de raíces complejas no reales, ya que si un complejo no real es raíz su complejo conjugado también lo es.

3. En consecuencia, el número de raíces reales de un polinomio de grado cuatro con coeficientes reales es siempre par.
4. Supongamos para empezar que $a = 0$, y podemos empezar dibujando la gráfica para diversos valores de b . Si no estaba claro desde el principio, vemos que todas las gráficas son iguales, la joroba doble de un dromedario invertida, y que al crecer b la gráfica “sube” respecto a los ejes.

Entonces, para valores suficientemente grandes de b no habrá ninguna raíz real y para valores muy negativos de b habrá dos raíces reales. Para valores intermedios debería haber cuatro raíces reales ya que el eje OX puede cortar a las dos jorobas.

5. Cuando $a = 0$ podemos tratar el problema usando que podemos escribir

$$p(x, 0, b) = (x^2 - 3)^2 - 9 + b,$$

y vemos inmediatamente que si $9 - b < 0$ no puede haber raíces reales y las cuatro raíces son números complejos no reales. Los casos restantes ($b \leq 9$) se pueden tratar de forma similar, resolviendo explícitamente la ecuación $p(x, 0, b) = 0$.

6. Supongamos ahora que $a \neq 0$ y vamos a tratar el caso particular en que $a = 8$. Si observamos cuidadosamente la gráfica para $a = 8$ y $b = 0$ vemos que una de las jorobas ha desaparecido y la parte de abajo de la gráfica se ha aplanado. ¿Por qué? Cuando estamos cerca del origen los dos primeros sumandos de $p(x, a, b)$ toman valores muy pequeños, si x es pequeño x^4 es mucho más pequeño, y el término $ax + b$ domina.
7. Para ver el motivo de la desaparición de una de las jorobas debemos calcular los máximos y mínimos de la función, es decir, debemos derivar e igualar a cero. Vemos entonces que lo que tiene de especial el valor $a = 8$ es que para ese valor y $x = 1$ se anulan la primera y la segunda derivadas. En la gráfica eso se ve como una zona casi horizontal cerca de $x = 1$.
8. En la gráfica también parece verse que la función $p(x, 8, 0)$ es creciente para $x \geq 0$. ¿Es esto verdad?
9. Debemos pensar que la desaparición de una de las jorobas implica que ahora sólo puede haber 2 raíces reales o bien ninguna. ¿Cómo se puede probar esta afirmación y para qué valores de b se obtendría cada uno de los casos?
10. ¿Puedes decir algo, acerca del número de raíces reales en función de a y b , si $a \neq 0$ y $a \neq \pm 8$?

Puedes ver una solución en la hoja [24-CAVAN-raices-reales.ipynb](#), y de la parte más matemática en este [archivo](#).

2.7.3. Algunos ejercicios más

Ejercicio 6. Comenzamos con el semicírculo de radio uno y centro el origen. Elegimos un ángulo $0 < \theta < \pi/2$ y representamos las rectas $x = \text{sen}(\theta)$ y $x = -\cos(\theta)$ de forma que junto con el eje $y = 0$ y la circunferencia encierran una región que llamamos A .

Llamamos B al complemento de A en el semicírculo. DETERMINAR el valor máximo, cuando θ varía, del cociente $F(\theta) := \text{Area}(A)/\text{Area}(B)$.

Ejercicio 7. Consideramos la parábola de ecuación $y = x^2$ y una circunferencia de centro $(0, a)$ y tal que es tangente⁹ a la parábola en dos puntos distintos. DETERMINAR los valores de a para los que tal circunferencia existe.

Ejercicio 8. Determinar el valor de m que hace que la ecuación

$$x^4 - (3m + 2)x^2 + m^2 = 0$$

tenga cuatro raíces reales en progresión aritmética (i.e. las raíces serían $x_0, x_0 + a, x_0 + 2a, x_0 + 3a$).

Ejercicio 9. Sea M un real muy grande. Demuestra que la ecuación $Mx = e^x$ tiene una solución, digamos w , única con $w > 1$. Estima el valor de w , usando como guía representaciones gráficas de $f(x) := e^x - Mx$, después de darle a M un valor suficientemente grande. ¿Podrías estimar el valor de M a partir del que se obtiene la solución w única? ¿Qué ocurre para valores de M más pequeños?

Ejercicio 10. Determina el valor mínimo de la constante $a > 1$ tal que siempre que $x \leq y$ se verifica

$$\frac{a + \text{sen}(x)}{a + \text{sen}(y)} \leq e^{y-x}.$$

Ejercicio 11. ¿Es mayor e^π que π^e ? A una pregunta así podríamos responder calculando los dos números reales, con cierta aproximación, y resulta que ciertamente es mayor e^π . Sin embargo, se trata de demostrarlo en la forma más convincente posible, sin usar el valor aproximado. Dicho de otra manera, nuestro argumento nos debe conducir a una demostración puramente matemática, para la que podemos usar como apoyo el ordenador (gráficas, derivadas, etc.).

Puedes ver algo de ayuda para los primeros dos ejercicios en este [archivo](#). Además, en la hoja [27-CAVAN-tangencias.ipynb](#) puedes ver una animación de la solución del segundo.

⁹Dos curvas son tangentes en un punto en que se cortan si tienen la misma recta tangente en ese punto.

2.7.4. Ejercicios de Álgebra Lineal

Antes de plantear una lista de ejercicios resolvemos, a modo de ejemplo, un problema de interpolación. Si bien la solución propuesta dista de ser la más eficiente, nos sirve como motivación a la manera de hacer que se pretende en este curso. En una hoja de Sage, tanto el enunciado como los comentarios en la solución, deberían aparecer en cuadros de texto.

Ejercicio 12. Método de coeficientes indeterminados. *Encontrar el polinomio de menor grado cuya gráfica pasa por los puntos*

$$(-2, 26), (-1, 4), (1, 8), (2, -2).$$

SOLUCIÓN.- Puesto que se tienen 4 puntos, se considera un polinomio general, de grado ≤ 3 : $P(x) = a_0 + a_1x + a_2x^2 + a_3x^3$.

Con las coordenadas de los 4 puntos dados, sustituyendo las abscisas e igualando a las respectivas ordenadas, se obtiene un sistema lineal de 4 ecuaciones con 4 incógnitas (los coeficientes, a_0, a_1, a_2, a_3 , del polinomio):

$$\begin{array}{ll} 1 * a_0 - 2 * a_1 + 4 * a_2 - 8 * a_3 = 26 & \text{punto } (-2, 26) \\ 1 * a_0 - 1 * a_1 + 1 * a_2 - 1 * a_3 = 4 & \text{punto } (-1, 4) \\ 1 * a_0 + 1 * a_1 + 1 * a_2 + 1 * a_3 = 8 & \text{punto } (1, 8) \\ 1 * a_0 + 2 * a_1 + 4 * a_2 + 8 * a_3 = -2 & \text{punto } (2, -2) \end{array}$$

El hecho de que los 4 puntos se encuentren en distintas verticales, asegura que el sistema es compatible determinado. Su matriz es

$$\begin{pmatrix} 1 & -2 & 4 & -8 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \end{pmatrix}$$

con inversa

$$\begin{pmatrix} -\frac{1}{6} & \frac{2}{3} & \frac{2}{3} & -\frac{1}{6} \\ \frac{1}{12} & -\frac{1}{3} & \frac{1}{3} & -\frac{1}{12} \\ \frac{1}{12} & -\frac{1}{6} & -\frac{1}{6} & \frac{1}{12} \\ -\frac{1}{12} & \frac{1}{6} & -\frac{1}{6} & \frac{1}{12} \end{pmatrix}$$

y la solución del sistema, obtenida multiplicando la matriz inversa por el vector columna de términos independientes, es el vector $(4, 5, 2, -3)$, es decir, el polinomio $4 + 5x + 2x^2 - 3x^3$. ■

Ejercicio 13. La suma de los n primeros enteros positivos, $1 + 2 + \cdots + n$, es un polinomio en n de grado 2: $\frac{1}{2}n^2 + \frac{1}{2}n$. La suma de sus cuadrados, $1^2 + 2^2 + \cdots + n^2$, es un polinomio de grado 3. En general, fijada la potencia, k , la suma $1^k + 2^k + 3^k + \cdots + n^k$ es un polinomio en n de grado $k + 1$ (¿POR QUÉ?).

Encontrar, usando interpolación como en el ejercicio anterior, una fórmula para la suma de los cubos de los primeros n enteros positivos.¹⁰

Podemos comprobar el resultado obtenido mediante las instrucciones

```
var('m k');sum(k^3,k,1,m)
```

que devuelve directamente el polinomio buscado, en m .

Ejercicio 14. Dada una matriz \mathbf{A} de tamaño $m \times n$ con entradas racionales, podemos definir una función $\Phi_{\mathbf{A}}$, entre espacios vectoriales de matrices, mediante

$$\begin{array}{ccc} \mathcal{M}_{n \times k} & \xrightarrow{\Phi_{\mathbf{A}}} & \mathcal{M}_{m \times k} \\ \mathbf{B} & \mapsto & \mathbf{A} \cdot \mathbf{B}, \end{array}$$

con $\mathcal{M}_{n \times k}$ ($\mathcal{M}_{m \times k}$) los espacios de matrices con n filas y k columnas (m filas y k columnas).

Cuando $k = 1$, la aplicación lineal $\Phi_{\mathbf{A}}$ es bien conocida: se trata de la obtenida al multiplicar vectores columna de n filas, colocados a la derecha de \mathbf{A} , por la propia matriz, y nos referimos a ella como la **aplicación lineal asociada** a la matriz \mathbf{A} . Sabemos que toda aplicación lineal $u : \mathbb{Q}^n \rightarrow \mathbb{Q}^m$ es la aplicación asociada a una matriz \mathbf{U} , a la que llamamos LA MATRIZ de la aplicación lineal.

1. Demostrar (completamente) que $\Phi_{\mathbf{A}}$ es, de hecho, una aplicación lineal.
2. Supongamos que la matriz \mathbf{A} es invertible, y, por tanto, cuadrada. ¿Será cierto que $\Phi_{\mathbf{A}}$ es necesariamente biyectiva (es decir, invertible)?
3. Supongamos que la aplicación lineal que corresponde a \mathbf{A} NO es inyectiva. ¿Es cierto que $\Phi_{\mathbf{A}}$ no puede ser inyectiva?
4. Supongamos que $m < n$, ¿es cierto que si la aplicación lineal que corresponde a \mathbf{A} es suprayectiva (es decir, de rango m) también será suprayectiva la aplicación lineal $\Phi_{\mathbf{A}}$? ¿Es cierta la afirmación recíproca?

¹⁰En el ejercicio 3, se pide una demostración, para este caso, de la afirmación de que estas sumas son polinomios en el número de sumandos.

5. Supongamos ahora que \mathbf{A} es 2×2 . En este caso, la aplicación lineal $\Phi_{\mathbf{A}}$ va del espacio de matrices $2 \times n$ en sí mismo.
- a) ¿Podrías calcular la matriz de $\Phi_{\mathbf{A}}$ en las bases estándar de los espacios de matrices (las matrices de esas bases tienen todas sus entradas 0 menos una que vale 1)?
 - b) Determina el núcleo de $\Phi_{\mathbf{A}}$, y discute los casos $\text{rango}(\mathbf{A}) = 0, 1, 2$.
 - c) Calcula una base del núcleo de $\Phi_{\mathbf{A}}$ en cada uno de los tres casos.

Capítulo 3

Estructuras de datos

Muchos de los programas que elaboramos sirven para transformar y analizar datos que creamos o recibimos. En este resumen se revisan las estructuras de datos que llamamos **lista**, **tupla**, **cadena de caracteres**, **conjunto** y **diccionario**. Una característica común a todos estos tipos es que son ITERABLES, es decir, que para todos ellos tiene sentido hacer un bucle **for** como

```
for item in <estructura_de_datos>:
```

```
.....
```

Nos referimos a estas estructura de datos con el nombre genérico de *contenedor*, o más precisamente *contenedor iterable*.

Otra cualidad común de los contenedores iterables es la de poder comprobar fácilmente si un dato está o no en ellas. Basta utilizar la partícula **in**, de manera que

```
dato in <estructura_de_datos>
```

adquiere valor **True** o **False**.

Por último, la función **len()** nos informa del número de datos en cualquiera de estas estructuras.

Existen, en Python, otros tipos de estructuras de datos más sofisticados pero en este curso nos bastará con los mencionados. Puedes consultar [esta página](#) para obtener más información.

3.1. Datos básicos: tipos

Algunos lenguajes de programación, por ejemplo *C*, obligan a declarar explícitamente de qué tipo es cada variable. Así por ejemplo, en *C* hay tipos como

1. **long** para almacenar enteros, que deben pertenecer al intervalo¹

$$[-2147483648, 2147483648]$$

. Cada entero, de tipo **long**, utiliza 32 bits en la memoria del ordenador.

2. **float** para números decimales, de hasta 8 cifras decimales, pertenecientes al intervalo

$$[1.175494351 \cdot 10^{-38}, 3.402823466 \cdot 10^{38}].$$

Cada decimal, de tipo **float**, utiliza 32 bits en la memoria del ordenador.

3. **double** para números decimales de mayor precisión, dieciseis cifras decimales, que los dados por **float**, concretamente, pertenecientes al intervalo

$$[2.2250738585072014 \cdot 10^{-308}, 1.7976931348623158 \cdot 10^{308}].$$

Cada decimal, de tipo **double**, utiliza 64 bits en la memoria del ordenador.

4. **char** en los que se almacenan los caracteres del código ASCII, las letras y símbolos de puntuación.

Cada carácter, tipo **char**, utiliza 8 bits en la memoria.

La declaración de tipos es importante porque hace posible la compilación del programa, es decir la conversión del código en un ejecutable que puede funcionar de manera totalmente independiente. Para ese funcionamiento independiente, en otro ordenador en el que no están necesariamente ni el código ni el compilador de *C*, es esencial que el ejecutable contenga toda la información sobre la gestión de la memoria RAM, que es lo que explica la necesidad de la declaración de tipos. El compilador traduce el código *C* a código máquina del procesador que tenga el ordenador que usamos, de forma que el ejecutable funcionará en cualquier otro ordenador con ese procesador.

¹Este rango depende del compilador de *C* utilizado, pero el indicado es el habitual.

En Sage, y en Python, no es necesario declarar los tipos de las variables debido a que el código no se compila, sino que se *interpreta*: el código se va traduciendo, por el *intérprete de Python*, a lenguaje máquina *sobre la marcha*, y es el intérprete el que se ocupa de la gestión de la memoria RAM. La ejecución de código en lenguajes interpretados es, en general, mucho más lenta que en lenguajes compilados, pero en cambio es mucho más cómodo programar en Python (Sage) que en *C*.

En Sage los datos tienen tipos, aunque no hay que declararlos, y se pueden producir *errores de tipo* (*Type error*). Por ejemplo, si intentamos factorizar un número decimal, se producirá un tal error.

Los objetos básicos en Sage, a partir de los que se construyen objetos complejos, son similares a los de *C*: enteros, decimales, caracteres. Dado el enfoque de Sage hacia las matemáticas, hay otros tipos de datos, números racionales o complejos, que también podemos considerar básicos.

Cuando nos encontramos con un error de tipo, al aplicar una función de Sage a un cierto objeto *A*, podemos evaluar `A.type()` o `type(A)`, lo que nos informará del tipo que tiene el objeto *A*. Analizando la documentación sobre la función podremos darnos cuenta de que esa función concreta no se puede aplicar a los objetos del tipo de *A*, y quizá deberemos cambiar la definición de *A*.

Hay dos tipos diferentes de enteros: los enteros de Python y los enteros de Sage. La única diferencia, en la práctica, es que no es posible aplicar a los enteros de Python los métodos disponibles para los enteros de Sage. Por ejemplo, los enteros de Sage tienen el método `.digits()`, que nos devuelve una lista ordenada de los dígitos del entero, mientras que si intentamos aplicar este método a un entero de Python obtenemos un error de tipo.

En este capítulo veremos la forma de crear estructuras de datos complejas, que contienen objetos básicos de alguno de los tres tipos, o bien otras estructuras de datos. Así, por ejemplo, podemos pensar una matriz como una lista que contiene listas de la misma longitud, que serán las filas de la matriz. Estas construcciones tienen, como veremos, algunas restricciones, y así, por ejemplo, no podemos crear un conjunto cuyos elementos sean listas.

3.2. Listas

En este curso USAREMOS SOBRE TODO LISTAS Y MATRICES como estructuras de datos, y las otras estructuras de datos servirán sobre todo en situaciones particulares en las que necesitaremos un contenedor más especializado.

1. Definimos una lista separando sus elementos por comas y delimitando el principio y el final de la lista con corchetes cuadrados:

```
L = [1,2,3,4,5]
```

```
type(L), len(L), L
```

```
(<type 'list'>, 5, [1, 2, 3, 4, 5])
```

El nombre `L` queda asignado a la lista `[1,2,3,4,5]`.

2. Podemos cambiar, en una lista L existente, el elemento con índice i reasignando su valor mediante `L[i]=\dots`. Por ejemplo `L[1]=7` transforma la lista L del apartado anterior en `[1, 7, 3, 4, 5]`.
3. Usaremos la notación *slice* para el acceso a sublistas. Si L es una lista que ya ha sido definida,
 - seleccionamos el elemento de índice j (o lugar $j+1$) en L con `L[j]`;²
 - seleccionamos con `L[j:k:d]`, la sublista formada por los elementos de índices $j, j+d, j+2d, \dots, j+\ell d < k$. Si no se especifica el tercer argumento, los *saltos* son de uno en uno. La ausencia de alguno de los dos primeros argumentos, manteniendo los dos puntos de separación, indica que su valor es el más extremo (primero o último).

```
L=[0,1,2,3,4,5,6,7,8,9,10,11,12]
```

```
print (L[3], L[1:5:2], L[6:10], L[11:], L[:4], L[::-3])
```

```
(3,[1, 3],[6, 7, 8, 9],[11, 12],[0, 1, 2, 3],[0, 3, 6, 9, 12])
```

4. El signo `+` concatena listas

```
[1,3,5]+[2,4,6]
```

```
[1, 3, 5, 2, 4, 6]
```

5. Se abrevia la concatenación k veces de una misma lista L con `k*L`.

²Esto se debe a que **Sage**, y **Python**, llaman elemento 0 de las listas al primero.

3.2.1. Métodos y funciones con listas

1. Al aplicar un método a un contenedor tipo lista, este cambia. Esta es una característica distintiva de una lista: es un contenedor de datos *mutable*.

- El constructor `list()` nos permite crear una copia exacta de una lista.

```
L=[1,2,3,4,5]
LL=list(L)
L.append(12)
L,LL
```

```
([1, 2, 3, 4, 5, 12], [1, 2, 3, 4, 5])
```

- Por ejemplo, `L.sort()` cambia la lista L de números enteros por la lista ordenada pero mantiene el nombre L. Si más adelante, en nuestro programa, necesitamos usar otra vez la lista L original es preciso generar una copia mediante `LL=list(L)`, o bien primero generar la copia LL y ordenar LL manteniendo L con su valor original. En particular, una asignación como `LL = L.sort()` no crea la lista LL aunque sí ordena L.
- En general, si se *alimenta* de cualquier otro contenedor iterable, se crea una lista con sus elementos. El siguiente ejemplo, que usa una cadena de caracteres, se entenderá mejor al leer la siguiente sección. Lo incluimos para ilustrar el uso de otro tipo de datos iterable:

```
Letras=list('ABCDE')
Letras
```

```
['A', 'B', 'C', 'D', 'E']
```

- La función `srange()` permite crear listas de **enteros de Sage**. En python se usa la función `range()`, que devuelve **enteros de Python**, a los que no son aplicables los métodos de Sage.
 - `srange(j,k,d)`: devuelve los números entre j (inclusive) y k (exclusive), pero contando de d en d elementos. A pesar de que el uso más extendido es con números enteros, los números j, k y d pueden ser enteros o no.

Abreviaturas:

- **srange**(k): devuelve **srange**(0,k,1). Si, en particular, k es un natural, devuelve los naturales entre 0 (inclusive) y k (exclusive); y si k es negativo, devuelve una lista vacía.
 - **srange**(j,k): devuelve la lista **srange**(j,k,1). Si, en particular, j y k son enteros, devuelve los enteros entre j (inclusive) hasta el anterior a k.
 - [a..b] es equivalente a **srange**(a,b+1).
- o El método **.reverse()** cambia la lista por la resultante de reordenar sus elementos dándoles completamente la vuelta.

```
L=[2,3,5,7,11]
LL=list(L)
L.reverse()
L, LL
```

```
([11, 7, 5, 3, 2], [2, 3, 5, 7, 11])
```

- o Con **.append()** se añade un nuevo elemento, y solo uno, al final de la lista. Como método, cambia la lista. Así, **L.append(5)** equivale a **L=L+[5]** (o **L+= [5]**). Para ampliar con más de un elemento, existe el método **.extend()**.
- o Si los elementos de la lista son comparables, se pueden ordenar, de menor a mayor, con el método **.sort()**.
- o **sum(<lista_numérica>)**³ calcula la suma de los elementos de la lista, que deben ser números.
- o Listamos, para finalizar, métodos referentes a un elemento concreto y una lista: el número de apariciones se averigua con el método **.count()**; información sobre la primera aparición la ofrece **.index()**; se puede borrar con métodos como **.pop()** o **.remove()**; y se añade en una posición determinada con **.insert()**. La ayuda interactiva, (tras el nombre del método y **tabulador**, nos amplía esta información.

3.2.2. Otras listas

Muchos métodos y funciones de sage devuelven listas como producto final de su evaluación, o, al menos, información fácilmente utilizable para generar una lista. Terminamos esta sección con varios de estos ejemplos.

³Esta notación, que usaremos frecuentemente, indica el tipo de objetos a los que podemos aplicar la función **sum()**, en este caso a los objetos de tipo **lista de números** y solo a tales listas. Cuando la aplicamos, por ejemplo **sum([1,2,3])**, no escribimos los angulitos sino solo una lista, [1,2,3] o el nombre, por ejemplo L, de una lista de números.

- *List comprehension*. Más que un método o función, es una manera abreviada de creación de listas. Por su especial sintaxis, en ocasiones es difícil de entender en una primera lectura. En general, se utiliza para generar listas aplicando cierta transformación sobre otra lista, o con los elementos de otro contenedor iterable. Así, por ejemplo

```
[ j^2 for j in [1..5] ]
```

tomará, uno a uno, los elementos de la lista [1,2,3,4,5] y generará la lista de sus cuadrados:

```
[1, 4, 9, 16, 25]
```

Se puede, también, filtrar los elementos del contenedor original. De nuevo nos adelantamos a un material posterior, aunque es fácil entender el siguiente ejemplo.

```
[ j^2 for j in [1..20] if j.is_prime() ]
```

```
[4, 9, 25, 49, 121, 169, 289, 361]
```

Volveremos sobre este asunto en la subsección 4.2.3.

- Dado un **entero de Sage**, k , el método $k.\text{digits}()$ devuelve una lista cuyos elementos son sus dígitos. Por ejemplo, $123.\text{digits}()$ devuelve la lista [3,2,1] (atención al orden). Por defecto los dígitos se consideran en la base 10, y una lista *sin ceros a la izquierda*. Otra base de numeración se indica como primer argumento, y un número de dígitos fijo, m , se indica asignando al parámetro `padto` dicho valor: `padto=m`.

```
L=123.digits(padto=5)
LL=111.digits(2,padto=7)
L.reverse(),LL.reverse()
L, LL
```

```
([0, 0, 1, 2, 3], [0, 1, 1, 0, 1, 1, 1, 1])
```

- Los enteros de Sage se pueden factorizar en primos con el método `.factor()`. El resultado no se muestra como una lista, pero en ocasiones es útil la lista de factores y potencias: el constructor `list()` es apropiado para este objetivo.


```
k=3888
print k.factor()
L=list(k.factor())
L
```

```
2^4 * 3^5
[(2, 4), (3, 5)]
```

3.2.3. Ejercicios con listas

Ejercicio 1. Dada la lista $L=[3,5,6,8,10,12]$, se pide:

- averiguar la posición del número 8;
- cambiar el valor 8 por 9, sin reescribir toda la lista;
- intercambiar 5 y 9;
- intercambiar cada valor por el que ocupa su posición simétrica, es decir, el primero con el último, el segundo con el penúltimo, ...
- crear la lista resultante de concatenar a la original, el resultado del apartado anterior.

Ejercicio 2. La orden `prime_range(100,1001)` genera la lista de los números primos entre 100 y 1000.

Se pide, siendo `primos=prime_range(100,1001)`:

- averiguar el primo que ocupa la posición central en `primos`;
- averiguar la posición, en `primos`, de 331 y 631;
- extraer, conociendo las posiciones anteriores, la sublista de primos entre 331 y 631 (ambos incluidos);
- extraer una sublista de primos que olvide los dos centrales de cada cuatro;
- (**) extraer una sublista de primos que olvide el tercero de cada tres.

3.3. Tuplas

Las “tuplas” son similares a las listas, la mayor diferencia es que no podemos asignar nuevo valor a los elementos de una tupla. Decimos que las tuplas son “inmutables”. Las listas no lo son porque podemos reasignar el valor de una entrada de la lista, $L[1] = 7$, y también un método puede modificar el valor de una lista, por ejemplo, `L.reverse()` cambia el valor de `L`.

- Definimos una tupla mediante $t = (1,2,3,4,5,6)$.
- Nos referimos a los elementos de una tupla con la notación *slice*, de la misma forma que a los de una lista, así: $t[0]$ es el elemento en la primera posición de la tupla, $t[1:5]$ es la tupla $(2,3,4,5)$; etc.
- Con el símbolo de la suma se concatenan tuplas: $t + (1,5)$ es la tupla $(1,2,3,4,5,6,1,5)$. Y con el del producto, se repite: $2*(1,5)$ devuelve la tupla $(1,5,1,5)$.
- A diferencia de las listas, no se puede cambiar un elemento de una tupla intentando reasignar su valor:

```
t=(1,2,3,4,5,6,7)
t[1]=3
```

`TypeError: 'tuple' object does not support item assignment`

Si queremos cambiar algún elemento de una tupla, tenemos que reasignar toda la tupla.

En ocasiones, esta será una tarea ingrata y preferiremos un camino más directo. Pensemos en una tupla con algunas decenas de elementos. Para el intento anterior, podríamos hacerlo con una sintaxis algo enrevesada:

```
t1,t2=t[:1],t[2:]    ##cortamos la tupla en dos,
##evitando el elemento a sustituir
t1+(3,)+t2    ## concatenamos intercalando la tupla (3,)
```

`(1, 3, 3, 4, 5, 6, 7)`

Es fácil comprobar que (3) no se interpreta como una tupla, pero $(3,)$ sí.

Otra opción es cambiar a un contenedor tipo lista, más manipulable, realizar los cambios y asignar el resultado a una tupla. Los constructores `list()` y `tuple()` son idóneos para este camino que se deja como ejercicio al lector.

- La función `len()`, aplicada a una tupla, nos devuelve su tamaño.
- Los métodos `.index()` y `.count()` se aplican de manera análoga al caso de listas.

3.3.1. Ejemplos con tuplas

Ejemplo 1. La composición `list(factor())`, aplicada a un natural, devuelve una lista de pares

```
factores=list(factor(600)); factores
```

```
[(2, 3), (3, 1), (5, 2)]
```

Se puede iterar sobre esta lista para extraer los pares

```
for k in factores: print k,
```

```
(2, 3) (3, 1) (5, 2)
```

o *desempaquetar* cada par, es decir extraer simultáneamente cada miembro de la pareja al iterar la lista

```
for a,b in factores: print a^b,
```

```
8 3 25
```

Ejemplo 2. La función `xgcd()` aplicada a dos enteros, a, b , devuelve una tupla con 3 valores: (d, u, v) . El primero es el máximo común divisor⁴, y se verifica la *identidad de Bézout*: $d = a \cdot u + b \cdot v$.

```
a,b=200,120
```

```
d,u,v=xgcd(a,b)
```

```
d,u,v,d-(a*u+b*v)
```

```
(40, -1, 2, 0)
```

En este curso usaremos mucho más listas que tuplas, ya que, esencialmente, realizan la misma función y las listas son más flexibles.

⁴*gcd* son las siglas de *greatest common divisor*. La función `gcd()` devuelve, sin más, el máximo común divisor; `xgcd()` es una versión extendida.

3.4. Cadenas de caracteres

Una cadena de caracteres es otro contenedor de datos inmutable, como las tuplas. Lo que contiene son caracteres de un alfabeto, por ejemplo el nuestro (*alfabeto latino*).

Dado que la CRIPTOGRAFÍA trata del enmascaramiento sistemático de un texto hasta hacerlo ilegible, pero recuperable conociendo la clave, es claro que en el capítulo 8, dedicado a ella usaremos sistemáticamente las funciones y métodos que se aplican a cadenas de caracteres.

- Una **cadena de caracteres** se forma concatenando caracteres de un alfabeto, habitualmente formado por letras, dígitos, y otros símbolos como los de puntuación.
- En el siguiente ejemplo se asigna a C una cadena

```
C='Esta es nuestra casa.'
```

Las comillas delimitan el inicio y el final de la cadena. En este caso es suficiente con comillas simples. Si se quieren incluir comillas simples entre los caracteres, se usarán comillas dobles⁵ o triples (tres sencillas consecutivas) para delimitar; forzosamente triples si han de incluir simples y dobles.

- **str**(k) convierte el entero *k* en una cadena de caracteres, de forma que **str**(123) da como resultado la cadena '123'.
- Las operaciones + (concatenación) y * (repetición) se utilizan y comportan como en el caso de listas.
- **len**(<cadena>) es el número de caracteres de la cadena.
- Para el acceso a subcadenas, se utiliza la notación *slice*.

```
frase='Recortando letras de una frase'
print (frase[3],frase[1:5:2],frase[6:10],frase[11:],frase[:4],frase[::3])
```

```
('o', 'eo', 'ando', 'letras de una frase', 'Reco', 'Roaoeadu a')
```

⁵Comillas dobles no son dos simples consecutivas, sino las que suelen estar en la tecla **2** de un teclado.

- `<cadena>.count(<cadena1>)` devuelve el número de veces que la `cadena1` aparece “textualmente” dentro de la `cadena`.

```
frase='Contando letras de una frase'
frase.count('e')
```

3

- `<cadena>.index(<cadena1>)` devuelve el lugar en que la `cadena1` aparece POR PRIMERA VEZ dentro de la `cadena`.

```
frase='Buscando letras en una frase'
frase.index('a')
```

4

- El método `.split()` trocea una cadena de caracteres, devolviendo una lista con las subcadenas resultantes. Si no hay argumento, se utiliza, por defecto, el espacio en blanco para trocear, eliminándose de la lista de subcadenas resultante. Si se indica al método una subcadena, se utiliza esta para recortar.

```
frase='Una_frase_para_trocear.
Por_defecto,_se_recorta_por_el_espacio_en_blanco.'
print frase.split('_')==frase.split()
palabras=frase.split('_')
subfrases=frase.split('.')
palabras; subfrases
```

```
True
['Una', 'frase', 'para', 'trocear.', 'Por', 'defecto,', 'se',
'recorta',
'por', 'el', 'espacio', 'en', 'blanco.']
['Una frase para trocear', ' Por defecto, se recorta por el espacio en
blanco', '']
```

- El efecto inverso al uso de `.split()` viene dado por el método `.join()`, que actúa sobre cadenas y espera un contenedor de cadenas; o la función `join()`, que actúa sobre un contenedor de cadenas. Un ejemplo, conectado con el anterior, nos sirve para entender el uso

```
'-'.join(palabras); join(palabras, ' | ')
```

```
'Una-frase-para-trocear.-Por-defecto,  
-se-recorta-por-el-espacio-en-blanco.'
```

```
'Una|frase|para|trocear.|Por|defecto,  
|se|recorta|por|el|espacio|en|blanco.'
```

De no especificarse el segundo argumento de la función `join()`, se utilizará un espacio en blanco simple como *pegamento*. Para concatenar cadenas, aparte del signo `+`, podemos usar `join` con la cadena *vacía* `''`

```
palabras[0]+palabras[1]; ''.join(palabras); join(palabras, '')
```

```
'Unafrase'
```

```
'Unafraseparatrocear.Pordefecto,serecortaporelespacioenblanco.'
```

```
'Unafraseparatrocear.Pordefecto,serecortaporelespacioenblanco.'
```

- El método `.find()` devuelve el primer índice en que aparece una subcadena en la cadena a que se aplica; si no aparece, devuelve `-1`. Se puede delimitar el inicio de la búsqueda, o el inicio y el final (de omitirse este, es el último índice de la cadena). En la cadena `frase` anterior aparece la subcadena `'or'` en 3 ocasiones. El siguiente código nos encuentra dónde:

```
L=len(frase)  
a=frase.find('or')  
b=frase.find('or',a+1,-2)  
c=frase.find('or',b+1)  
a, b, c, frase.find('or',c+1)
```

```
(25, 43, 49, -1)
```

Sugerencia: Pulsar el `tabulador` tras escribir un punto detrás del nombre de una variable que contenga una cadena de caracteres, `<cadena>`. `+ tabulador`, para encontrar los métodos aplicables a cadenas de caracteres.

3.4.1. Ejercicios

Ejercicio 3. *Considérese el número 1000! (**factorial**(1000)):*

- *¿en cuántos ceros acaba?*
- *¿Se encuentra el número 666 entre sus subcadenas? En caso afirmativo, localizar (encontrar los índices de) todas las apariciones.*
- *Encontrar la subcadena más larga de doses consecutivos. Mostrarla con los dos dígitos que la rodean.*

Ejercicio 4. *Si se aplica la función **sum**() a una lista numérica, nos devuelve la suma de todos sus elementos. En particular, la composición **sum**(k.**digits**()), para k un variable entera, nos devuelve la suma de sus dígitos (en base 10).*

- *Calcular, con la composición **sum**(k.**digits**()), la suma de los dígitos del número k=**factorial**(1000).*
- *Calcular la misma suma sin utilizar el método **.digits**()*.

Sugerencia: considerar la cadena `digitos='0123456789'`, en la que `digitos[0]='0'`, `digitos[1]='1'`, ..., `digitos[9]='9'`. Sumar los elementos de la lista

$$[j * (\text{ veces que aparece } j \text{ en } 1000!) : \text{ con } j = 1, 2, 3, 4, 5, 6, 7, 8, 9].$$

3.5. Conjuntos

Los conjuntos en SAGE se comportan y tienen, esencialmente, las mismas operaciones que los conjuntos en Matemáticas. La mayor ventaja que tienen sobre las listas es que, por su propia definición como conjunto, suprimen las repeticiones de elementos. Así `[1, 2, 1]` tiene 3 elementos (y están indexados), pero `{1, 2, 1}` es lo mismo que `{1,2}` y tiene solo 2 elementos.

Así podemos usar conjuntos para suprimir repeticiones en una lista: transformamos la lista en conjunto y el conjunto resultante en lista.

- Se utilizan las llaves para delimitar un conjunto:

```
A = {1,2,1}
type(A), A
```

```
(<type 'set'>, set([1, 2]))
```

- Aunque los conjuntos son mutables, sus elementos han de ser objetos inmutables. Así, no podemos crear conjuntos con conjuntos o listas entre sus elementos.

```
A={{1,2},{3}}
```

```
TypeError: unhashable type: 'set'
```

```
A={1,2}
```

```
TypeError: unhashable type: 'list'
```

- Otra manera de crear conjuntos es con el constructor `set()`, que toma los elementos de cualquier contenedor

```
l,t,s=[1,2,1],[3,2,3],'Hola gente'
set(l); set(t); set(s)
```

```
set([1, 2])
set([2, 3])
set(['a', ' ', 'e', 'g', 'H', 'l', 'o', 'n', 't'])
```

Nota: La orden `A=set()` crea un conjunto sin elementos, el *conjunto vacío*.

- Los elementos de un conjunto no están ordenados ni indexados: si `A` es un conjunto `A[0]` no es nada.
- La instrucción `1 in A` devuelve **True** si el 1 es uno de los elementos del conjunto `A`.
- Si `A` y `B` son dos conjuntos, la unión de dos conjuntos se obtiene con `A | B`, la intersección con `A & B` y la diferencia con `A - B`. La comparación `A <= B` devuelve **True** si todos los elementos de `A` están en `B`.
- Como para los contenedores ya estudiados, `len(A)` es el número de elementos de un conjunto.
- Añadimos un elemento, por ejemplo el 1, a un conjunto mediante `A.add(1)`, y lo suprimimos con `A.remove(1)`. Si se quieren añadir más elementos, contenidos en cualquier otro contenedor, basta aplicar el método `.update()`


```
A={-1,-2}
l=srange(100)
A.update(l)
len(A)
```

102

Sugerencia: Averiguar el uso de otros métodos como: `.pop()`, `.union()`, `.intersection()`, ...

3.5.1. Ejercicios

Ejercicio 5. A partir de la cadena de caracteres

```
texto='Omnes homines, qui sese student praestare ceteris animalibus,
summa ope niti decet ne uitam silentio transeant ueluti pecora, quae
natura prona atque uentri oboedientia finxit.'
```

extraer la lista de los caracteres del alfabeto utilizados, sin repeticiones, sin distinguir mayúsculas de minúsculas y ordenada alfabéticamente.

Sugerencia: La composición `list(set())` aplicada a una lista, genera una lista con los elementos de la original sin repeticiones, ¿por qué?

Ejercicio 6. Máximo común divisor.

Sin utilizar los métodos `.divisors()` ni la función `max()`, elabora código que, a partir de dos números a y b , calcule:

- El conjunto $\text{Div}_a = \{k \in \mathbb{N} : k|a\}$
- El conjunto $\text{Div}_b = \{k \in \mathbb{N} : k|b\}$
- El conjunto $\text{Div}_{a,b} = \{k \in \mathbb{N} : k|a \text{ y } k|b\}$.

Una vez se tenga el conjunto de los divisores comunes, encontrar el mayor de ellos.

Ejercicio 7. Mínimo común múltiplo.

El mínimo común múltiplo, m , de dos números, a y b , es menor o igual que su producto: $m \leq a \cdot b$. Sin utilizar la función `min()`, elabora código que, a partir de dos números a y b , calcule:

- El conjunto $\text{Mult}_{a(b)} = \{k \in \mathbb{N} : a|k \text{ y } k < a \cdot b\}$
- El conjunto $\text{Mult}_{b(a)} = \{k \in \mathbb{N} : b|k \text{ y } k < a \cdot b\}$
- El conjunto $\text{Mult}_{a,b} = \{k \in \mathbb{N} : a|k, b|k \text{ y } k < a \cdot b\}$

Una vez se tenga este subconjunto de los múltiplos comunes, encontrar el menor de ellos.

Ejercicio 8. Dada una lista de números enteros, construye un conjunto con los factores primos de todos los números de la lista.

Indicación: Usa `list(k.factor())` para obtener los factores primos.

Estos ejercicios, y los de las páginas 61 y 67, son importantes porque vamos a estar usando manipulaciones de estructuras de datos, sobre todo listas y cadenas de caracteres, durante todo el curso. Soluciones en la hoja de Sage [31-ESTRD-ejercicios-sol.ipynb](#).

3.6. Diccionarios

Los elementos de una lista L están indexados por los enteros entre 0 y `len(L)-1` y podemos localizar cualquier elemento si conocemos su orden en la lista. Los diccionarios son parecidos a las listas en que los elementos están indexados, pero el conjunto de índices es arbitrario y adaptado al problema que queremos resolver.

- Definimos un diccionario mediante

```
diccionario = {clave1:valor1, clave2:valor2, ...}
```

Las claves son los identificadores de las entradas del diccionario y, por tanto, han de ser todas diferentes. En general, la información que contiene el diccionario reside en los pares clave:valor. Por ejemplo, un diccionario puede contener la información `usuario:contraseña` para cada uno de los usuarios de una red de ordenadores.

- o Una vez creado el diccionario, recuperamos la información sobre el valor correspondiente a una clave con la instrucción `diccionario[clave]`.

```
granja={'vacas':3, 'gallinas':10, 'ovejas':112}
granja['ovejas']
```

112

- o Definimos una nueva entrada, o cambiamos su valor, mediante

`diccionario[clave]=valor,`

y suprimimos una entrada con **del** `diccionario[clave]`.

```
granja['vacas']+=1 ## Se adquieren una vaca...
granja['caballos']=2 ## ... y dos caballos.
del granja['gallinas'] ## Se venden todas las gallinas.
show(granja)
```

`{vacas : 4, ovejas : 112, caballos : 2}`

- o Podemos crear un diccionario vacío con el constructor **dict()**. Si pasamos al constructor una *lista de pares*, este creará un diccionario con claves los primeros objetos de cada par, y valores los segundos.

```
Cuadrados_y_cubos=dict([(j,(j^2,j^3)) for j in [1..6]])
show(Cuadrados_y_cubos)
```

`{1 : (1, 1), 2 : (4, 8), 3 : (9, 27), 4 : (16, 64), 5 : (25, 125), 6 : (36, 216)}`

Nota: Un buen constructor de listas de pares es **zip()**. Si se tienen dos listas, L1 y L2, la lista **zip(L1,L2)** está formada por las parejas (L1[j],L2[j]) para j=0,1,..., el menor índice final posible:

```
L1, L2=[1..15], [10..16]
DD=dict(zip(L1,L2))
print DD
```

```
{1: 10, 2: 11, 3: 12, 4: 13, 5: 14, 6: 15, 7: 16}
```

- Los métodos `.keys()` y `.values()` producen listas con las claves y los valores, respectivamente, en el diccionario. El método `.items()` devuelve la lista de pares (clave,valor).
- La instrucción `x in diccionario` devuelve **True** si `x` es una de las claves del diccionario.

Los diccionarios sirven para ESTRUCTURAR LA INFORMACIÓN, haciéndola mucho más accesible. En el capítulo 8, dedicado a la criptografía, veremos un ejemplo muy próximo a la noción de “diccionario como un libro con palabras ordenadas alfabéticamente”: queremos decidir de forma automática si un texto dado pertenece a un idioma y disponemos de un archivo que contiene unas 120.000 palabras en ese idioma, una en cada línea.

Podríamos crear una lista con entradas las palabras del archivo, pero es mucho más eficiente estructurar la información como un diccionario. Buscar una palabra en una tal lista requiere recorrer la lista hasta que la encontremos, y sería equivalente a buscar una palabra en un diccionario de papel comparándola con cada palabra del diccionario empezando por la primera.

En lugar de usar una lista, creamos un diccionario con claves tripletas de caracteres y cada tripleta toma como valor la lista de todas las palabras en el archivo que comienzan exactamente por esa tripleta. De esta forma conseguimos que los valores en el diccionario sean listas de longitud moderada, y resulta mucho más fácil averiguar si una cierta palabra está o no en el diccionario.

Puedes consultar la hoja de Sage [41-PROGR-diccionario.ipynb](#) para ver una implementación de esta idea, aunque ES NECESARIO HABER VISTO EL CAPÍTULO SIGUIENTE, dedicado a la programación, para entenderla.

Además, se discute en esa hoja la ventaja que se obtiene al usar diccionarios, siempre que las longitudes de los valores estén equilibradas, frente a listas. Este tipo de discusión se repetirá a lo largo del curso, ya que repetidas veces deberemos comparar, mediante *experimentos bien elegidos* los méritos o deméritos de diversos métodos para realizar los mismos cálculos.

3.7. Conversiones

En ocasiones necesitamos convertir los datos contenidos en una cierta estructura a otra, debido, esencialmente, a que la nueva estructura admite métodos que se adaptan mejor a las manipulaciones con los datos que pretendemos hacer.

1. TUPLA A LISTA O CONJUNTO: Para una tupla T

```
list(T)
set(T)
```

2. LISTA A TUPLA: **tuple**(L).

3. CADENA A LISTA:

```
C = 'abc'
list(C)
```

```
['a', 'b', 'c']
```

4. ¿LISTA A CADENA?:

```
C = 'abc'
str(list(C))
```

```
['a', 'b', 'c']
```

En general esto NO es lo que queremos.

5. LISTA A CADENA:

```
C = 'abc'
join(list(C),sep="")
```

```
'abc'
```

6. LISTA A CONJUNTO: para una lista L , **set**(L). Suprime repeticiones en la lista.

7. CONJUNTO A LISTA: para un conjunto A , **list**(conjunto).

8. DICCIONARIO A LISTA DE PARES: Para un diccionario D , **D.items**().

9. LISTA DE PARES A DICCIONARIO: Este pequeño programa, con un bucle **for**, realiza la conversión.

```
def convert_list_dict(L):  
    dict = {}  
    for item in L:  
        dict[item[0]]=item[1]  
    return dict
```

10. Dadas dos listas, L1 y L2, de la misma longitud podemos formar una lista de pares mediante **zip**(L1,L2), y transformar esta lista en diccionario mediante la función del apartado anterior.

Capítulo 4

Técnicas de programación

El contenido de este capítulo es fácil de describir: un programa consiste esencialmente en bloques **for**, que repiten un número de veces un grupo de instrucciones, y bloques **if**, que bifurcan la ejecución del código según se cumplan o no determinadas condiciones booleanas: si se cumple una condición ejecutan un bloque de instrucciones y si no se cumple otro distinto.

Se describe la sintaxis de ambos tipos de bloque y se muestra mediante ejemplos sencillos su uso, junto con otros métodos como la recursión y los bloques **while**.

Como se indicó en el prólogo, y nunca conseguiremos repetirlo un número suficiente de veces, la única forma conocida de aprender a programar es *programando*, y no es algo que se pueda asimilar en la última semana antes del examen final.

Por otra parte, la programación en algún lenguaje de alto nivel forma ya parte importante de la formación que se espera de un graduado en matemáticas, y, les guste mucho o no, es en lo que terminan trabajando muchos de nuestros graduados.

4.1. Funciones

En esta sección, y en otros lugares a lo largo de este texto, la palabra *función* designa un programa que recibe unos argumentos y devuelve, después de algunos cálculos, un resultado. Es claro que una función en este sentido define, si el conjunto de argumentos no es vacío, una *función matemática* del conjunto de todos los posibles argumentos al de todos los posibles resultados. Según el contexto se puede decidir si la palabra *función* se refiere a un programa o a una función matemática.

La sintaxis para definir funciones es:

```
def nombre_funcion(arg_1,arg_2,...,arg_n):
    '''Comentarios sobre la funcion'''
    instruccion_1
    instruccion_2
    etc.
return res_1,res_2,...,res_m
```

1. El código escrito en *Python* utiliza el sangrado de las líneas para indicar los distintos bloques de código. Otros lenguajes de programación, por ejemplo C, utilizan llaves para obtener el mismo resultado. Esto hace que, en general, sea más fácil leer código escrito en Python.

No es difícil acostumbrarse a “ver” los errores de sangrado, y el intérprete ayuda produciendo un *Indentation Error* cuando intentamos ejecutar código mal sangrado. Además, pinchando con el cursor a la izquierda del mensaje de error nos muestra el lugar aproximado donde ha encontrado el error mediante un angulito en la línea de debajo.

2. Los nombres `arg_j` y `res_i`, así como el de la función, `nombre_funcion`, son genéricos. Es una buena costumbre el utilizar nombres que sean lo más descriptivos posible. Así por ejemplo, si un resultado va a ser el cociente de una división es bueno utilizar como nombre `cociente` o `coct` en lugar de `res_3`.
3. Los comentarios son opcionales, pero ayudan a entender mejor el código cuando se relee pasado un tiempo, y también cuando lo leen personas diferentes a su autor. Es por tanto una buena práctica, aunque por pereza muchas veces no los escribimos.
4. Las instrucciones que forman el programa indican cómo calcular `res_i` utilizando los argumentos de la función `arg_j` y otras funciones ya definidas dentro de Sage o por nosotros.

En general, debe haber líneas en el código el tipo `res_i=...` que definan todos los valores que debe devolver `return`. Recuerdese (ver página 25) que una línea como estas asigna un valor, lo que está a la derecha del igual, a una *variable* de nombre `res_i`.

Si la función debe devolver un valor que no ha sido calculado se produce necesariamente un error.

5. Dentro de otra función, por ejemplo `nf2(arg_1, arg_2, ..., arg_k)` podemos *llamar* a la función `nombre_funcion` mediante una línea como

```
res_1, res_2, ..., res_m = nombre_funcion(arg_1, arg_2, ..., arg_n)
```

convenientemente sangrada. En las líneas anteriores, incluyendo la posibilidad de que algunos de esos valores sean argumentos de la propia función `nf2`, debe estar definido el valor de todos los argumentos de la función, y si no es así se producirá un error al interpretar el código.

Esta línea, si llega a ejecutarse sin errores, asigna un valor a todas las variables `res_i`.

6. De esta manera podemos dividir nuestra tarea en varias tareas simples, definidas cada una de ellas en una función propia, y ejecutar la tarea principal en una función que va *llamando* a las funciones auxiliares.

Se llama a esta técnica *programación estructurada*, y la usaremos ampliamente a lo largo del curso. Usándola es mucho más fácil escribir, leer o **depurar el código**.

4.2. Control del flujo

4.2.1. Bucles for

1. Un bucle **for** es una manera de repetir (“iterar”) la ejecución de un bloque de código un NÚMERO DE VECES DADO.
2. La sintaxis de un **for** es

```
for <elemento> in <contenedor>:
    instruccion 1
    instruccion 2
    etc ...
```

3. Este bucle empieza asignando a `<elemento>` el valor que ocupa la primera posición¹ en el `<contenedor>`, y termina para el valor en la última posición, `len(contenedor) - 1`. Por tanto, repite el bloque de instrucciones `len(contenedor)` veces. Así,

¹En los conjuntos y diccionarios los elementos no están explícitamente ordenados, y el bucle los recorre de acuerdo a la forma en que están almacenados en la memoria de la máquina. Las listas, tuplas y cadenas de caracteres están explícitamente ordenadas y el bucle las recorre en el orden en que las hemos definido.

por ejemplo, un bucle con línea de entrada **for j in srange(10)**: asignará a j los enteros de sage 0, 1, 2, ..., 9, y repetirá su bloque de instrucciones 10 veces.

4. Las instrucciones del bloque DEBEN ESTAR TODAS ALINEADAS y “sangradas” respecto al **for**.
5. Un bloque de código puede tener un subbloque, por ejemplo una de las instrucciones de un bloque puede ser un bloque **for** con una serie de instrucciones que forman el subbloque. En ese caso, el **for** que determina el subbloque está alineado con todas las instrucciones del bloque **for** inicial, y todas las instrucciones del subbloque están alineadas entre sí y sangradas respecto a las del primer bloque. La estructura sería:

```
for <elemento> in <contenedor>:
    instruccion 1
    instruccion 2
    for <elemento2> in <contenedor2>:
        instruccion 3
        instruccion 4
        .....
    instruccion m
    .....
```

Este bucle doble se ejecuta en la forma natural: para cada valor de <elemento> se ejecutan en orden instruccion 1, instruccion 2, y se entra en el segundo bucle, que se ejecuta completo, antes de poder continuar con la instruccion m y las que vengan a continuación.

Un ejemplo típico de un bucle doble es el que nos sirve para recorrer los elementos de una matriz $m \times m$:

```
def matriz_hilbert(m):
    A = matrix(QQ,m,m,[0]*m^2)
    for i in srange(m):
        for j in srange(m):
            A[i,j] = 1/(i+j+1)
    return A
```

6. En ocasiones, cada valor sobre el que itera el **for** es una variable que se usa explícitamente en las instrucciones del bloque:

```
cuadrados=[] ## iniciamos una lista vacía
for j in srange(20): ## 20 iteraciones
    cuadrados.append(j^2) ## actualizamos la lista cuadrados
cuadrados[1::2] ## listamos los de los impares

[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

Pero no es obligatorio, como vemos en el siguiente código² que sirve para calcular la suma de los $K = 100\,000$ primeros términos de una progresión aritmética:

```
a,d,K=3,5,10^5
suma=a
for j in xrange(K):
    a=a+d
    suma+=a
print suma

25000550003
```

7. EJEMPLO:

Programamos mediante un **for** (*iterativamente*) el cálculo del término n -ésimo de la sucesión de Fibonacci, definida mediante $F_0 := 0$, $F_1 := 1$, $F_n := F_{n-1} + F_{n-2}$ para $n \geq 2$:

```
def fibon(m):
    p,q = 1,0
    for j in xrange(m):
        p,q = q,p+q
    return q
```

²En ocasiones el rango aparece como **xrange()** en lugar de **srange()**. La diferencia fundamental es que el segundo rango genera una lista de enteros y luego la recorre, y el primero no genera la lista y va aumentando el valor del contador en cada vuelta. Para rangos grandes la segunda forma de la instrucción genera una lista enorme que puede saturar la memoria RAM de la máquina.

Cuando empezamos el bucle, el par (p, q) vale $(1, 0)$, y en cada vuelta sus valores se sustituyen simultáneamente por $(q, p + q)$. Al final, la función devuelve el último valor calculado, es decir, el valor de q cuando ya el **for** ha dado todas las vueltas que debe. En particular, `fibon(0)` devuelve 0, ya que `xsrangle(0)` es una lista vacía, de manera que no se entra ni una sola vez al bloque de instrucciones.

El procedimiento es muy eficiente, en el uso de la RAM, porque ni se genera una lista que recorrer ni se almacenan los términos de la sucesión. Como indica la definición de la sucesión de Fibonacci, en cada vuelta del bucle, no necesita acordarse de todos los valores anteriores de q , solo de los dos últimos.

Sage cuenta con una instrucción `fibonacci()` que realiza el mismo cálculo que la que acabamos de definir. ¿Es más eficiente que la nuestra?

4.2.2. Contadores y acumuladores

Supongamos que queremos calcular la suma de una gran cantidad de números enteros definidos mediante una cierta propiedad, por ejemplo *la suma de los primeros cien mil primos*.

Una posibilidad es generar una lista L que los contenga a todos y después aplicar la función `sum(L)`, pero si no nos interesa saber cuáles son los primos sino únicamente su suma es claro que esta forma de calcularla no es muy eficiente: estamos creando una estructura de datos enorme, la lista L , en la memoria de la máquina que en realidad no necesitamos.

En lugar del enfoque anterior podemos usar un ACUMULADOR, que simplemente es una variable que según vamos encontrando enteros que satisfacen el criterio los sumamos al valor del acumulador. Ya hemos visto un ejemplo al sumar los términos de una progresión aritmética un poco más arriba.

El código para la suma de los primeros N primos sería

```
def suma_primos(N):
    suma = 2
    primo = 2
    for j in xsrange(N-1):
        primo = next_prime(primo)
        suma += primo ##Equivale a suma = suma+primo
    return suma
```

y conviene definir otra función (EJERCICIO) que obtenga la suma creando la lista de todos los primos y luego sumando. Podemos registrar el tiempo que dura el cálculo comenzando la línea con **time**, como en **time** suma_primos(10^5), y, entonces, comparar el tiempo que tarda el código con acumulador con el que tarda el código que genera la lista.

La idea de un *contador* es similar: queremos *contar el número de veces que “algo” ocurre*, y en lugar de generar un contenedor C con todos los elementos que cumplen las condiciones requeridas para luego usar **len**(C), podemos usar una variable **cont** que inicialmente vale 0 e incrementa su valor en una unidad cada vez que encontramos un elemento que cumple las condiciones. Usaremos contadores profusamente en el capítulo 11.

Para implementar un contador podemos usar una estructura como

```
cont = 0
for int in srange(N):
    ....
    if condicion:
        ....
        cont += 1
return cont
```

Para poder escribir un programa así debemos conocer *a priori* el número N , el número de vueltas del bucle **for**, y si no lo conocemos **debemos intentar usar un bucle while**. La **condicion** que aparece en el **if** expresa la definición de los objetos que estamos contando, ya que el contador únicamente se incrementa cuando se cumple la condición.

Es claro que, si es posible, es más eficiente usar contadores y acumuladores que generar grandes estructuras de datos que en muchos casos no necesitamos. Como veremos en la subsección sobre *iteradores*, esa eficiencia, sobre todo en el consumo de RAM, mejora cuando usamos *iteradores* en lugar listas (**xrange**(N) en lugar de **srange**(N)).

4.2.3. Otra sintaxis para los bucles

En Python existe otra manera, muy concisa, de ejecutar bucles **for**: la instrucción

```
[f(x) for x in L]
```

produce una lista cuyas entradas son los valores de la función, previamente definida, $f(x)$ obtenidos al recorrer x la lista L . Además se puede filtrar la lista resultante con un **if**:

```
[f(x) for x in L if Condicion]
```

que solo calcula y se queda con los $f(x)$ cuando se cumple la condición booleana `Condicion`.

Un ejemplo típico sería

```
[m for m in srange(2,10^6) if is_prime(m)]
```

Es fácil traducir esta sintaxis a la estándar, y la única ventaja de ésta es la concisión. Si nos acostumbramos a esta sintaxis concisa corremos el peligro de usarla en situaciones en las que no necesitamos la lista resultante sino únicamente su longitud, su suma u otra característica. En esos casos es mucho más eficiente, como se indicó en la subsección anterior, usar contadores o acumuladores dentro de bucles estándar.

También es posible, como se indica al tratar de los *iteradores*, convertir esta sintaxis en un *generador* que va produciendo los elementos de la lista sin crear en memoria la lista completa.

4.2.4. Otra más

Otra forma concisa de escribir cierta clase de bucles es mediante `map(f,L)`, con f una función cualquiera y L una lista de sus posibles argumentos. El resultado es la lista `[f(item) for item in L]`, es decir, `map` aplica la función f a cada uno de los elementos de la lista L .

La función f puede ser una función predefinida de Sage a la que llamamos por su nombre (`sin`, `cos`, `exp`, `log`, `sqrt`, etc.) o una función definida por nosotros mediante

```
def f(x):
    return .....
```

Es claro que el resultado de `map(f,L)` es equivalente al programa

```
def map2(f,L):
    LL = []
    for item in L:
        LL.append(f(item))
    return LL
```

de forma que esencialmente se trata de una sintaxis cómoda para un bucle `for`.

Usaremos esta forma del bucle `for` en el capítulo 8, dedicado a la criptografía, para modificar textos aplicándoles una función que cambia unos caracteres por otros.

4.2.5. Los bloques **if** <condicion>:

1. Un **if** sirve para *ramificar la ejecución de un programa*: cada uno de las partes del **if** define un camino alternativo y el programa entra o no según se verifique o no una condición.

2. La sintaxis del **if** es:

```

if <condicion 1>:
    instruccion 1
    instruccion 2
    etc ...
elif <condicion 2>:
    instruccion 3
    instruccion 4
    etc ...
    etc ...
else:
    instruccion 5
    instruccion 6
    etc ...

```

3. Decimos que una *expresión o función es booleana* si cuando se ejecuta devuelve un valor de *verdadero o falso* (**True** o **False**) (ver la subsección 2.3.2). Ejemplos típicos son los operadores de comparación

a) $A == B$, es decir, A es idéntica a B que se puede aplicar tanto a números como a estructuras de datos.

b) $A < B$, $A <= B$, $A > B$, $A >= B$ que se aplica a objetos para los que hay un orden natural³

```
[3 < 2, 'a' < 'b', 'casa' > 'abeto', set([1,2]) >= set([1]) ]
```

```
[False, True, True, True]
```

³Averiguar el comportamiento con los números complejos.

- c) Muchas funciones predefinidas en Sage son booleanas, y en particular todas las que tienen un nombre de la forma `is_...`, como por ejemplo `is_prime(m)` que devuelve `True` si m es primo y `False` si no lo es. Podemos ver todas las funciones predefinidas cuyo nombre empieza por `is` escribiendo `is` en una celda y pulsando la tecla del tabulador.
4. Las diversas condiciones deben ser *booleanas* y puede haber tantas líneas `elif` como queramos.
 5. La ejecución del programa *entra a ejecutar el bloque situado debajo del `if`, `elif` o `else`* la primera vez que se cumple una de las condiciones, consideradas en orden descendente en el código. Cada vez que se ejecuta un `if` únicamente se ejecuta uno de los bloques.
 6. No es estrictamente necesario que las condiciones sean *disjuntas*, y si no lo son se aplica lo indicado en el apartado anterior. Sin embargo, es conveniente, porque es más claro cómo se ejecuta todo el bloque `if`, intentar que lo sean.
 7. La última parte, el `else` es opcional y lo usamos si queremos indicar lo que debe hacer el programa en el caso en que no se cumpla ninguna de las condiciones.

4.2.6. Bucles `while`

1. Los bucles `while` son similares a los `for`, pero los usamos cuando no sabemos, a priori, cuantas vueltas debe dar el bucle, aunque estamos seguros de que acabará.
2. Su sintaxis es:

```
while <condición>:  
    instrucción 1  
    instrucción 2  
    etc ...  
    actualización de la condición
```

No es necesario que la ‘actualización de la condición’ esté al final, pero sí es imperativo que aparezca.

3. El bucle se sigue ejecutando mientras la condición, que debe ser una expresión booleana, es cierta (valor de verdad `True`).

4. Un bucle como **while** $2 > 1$: ... es, en principio, un bucle infinito que no puede terminar ni producir un resultado salvo que haya un **break** que lo pare. Cuando, dentro de la ejecución de un bucle, el programa llega a una línea con un **break** el bucle termina y el programa sigue ejecutándose a continuación del bucle.

Siempre hay que tener mucho cuidado con los bucles **while** infinitos ya que, aparte de no producir ningún resultado, frecuentemente cuelgan la máquina.

5. En la condición debe haber una variable cuyo valor actualizamos, normalmente al final, dentro del bloque de instrucciones. Habitualmente, cada vuelta del bucle nos acerca más al momento en que la condición se hace falsa y el bucle se para.
6. EJEMPLOS:

- a) *Encontrar los dígitos cuya posición en la expresión decimal del factorial de un entero m (calculado mediante la instrucción **factorial**(m)) es un número de Fibonacci.*

```
def gen_subcadena(m):
    C = str(factorial(m)) #Convertimos el factorial de m en una cadena
    K = len(C)
    C1 = "" #Contendrá la solución
    j = 1
    while fibonacci(j) < K:
        C1 = C1+C[fibonacci(j)-1] #Añadimos a C1 un dígito cuya
posición es un número de Fibonacci
        j += 1 #Incrementamos j para que el bucle NO sea infinito
    return C1
```

Usar un **while** es muy conveniente porque no sabemos *a priori* cuál va a ser el primer número en la sucesión de Fibonacci que supere K . Podríamos intentar resolver primero el problema (matemático) consistente en calcular, dado un entero K , el mayor número de Fibonacci menor que K .

- b) El programa que sigue es EQUIVALENTE al anterior, aunque tiene más líneas, y es igual de eficiente gracias a que usa **xrange**(): en cuanto **fibon**(j) supera a K , el programa entra en el **else** y la instrucción **break** para el bucle **for**. Entonces, no importa que inicialmente el bucle esté definido para un j que puede llegar a $K - 1$ porque, de hecho, nunca llega y se para el bucle mucho antes. Acerca de la instrucción **xrange**() puede verse la nota a pie de página en el punto 5 de la subsección 4.2.1, o, en mucho mayor detalle, en la sección 5.5.2.1.

```
def gen_subcadena2(m):
    C = str(factorial(m))
    K = len(C)
    C1 = ""
    for j in xrange(1,K):
        Fj = fibonacci(j)
        if Fj < N:
            C1 = C1+C[Fj-1]
        else:
            break
    return C1
```

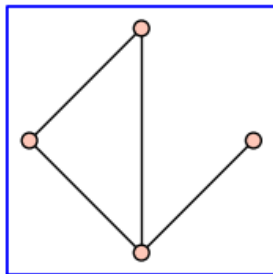
- c) Lee cuidadosamente los dos programas anteriores, y trata de entender que la respuesta que producen es realmente una solución del problema. Para eso puede ser conveniente hacer que la respuesta contenga más información, por ejemplo, que al lado de cada dígito calculado aparezca, entre paréntesis, el lugar que ocupa en la cadena C.
- d) Este ejemplo muestra que la combinación **for+if+break** puede ser equivalente a **while**. ¿Es siempre así? ¿De qué depende el que sean equivalentes?
- e) En ocasiones queremos usar un contador incrementado en un bucle del que no sabemos *a priori* cuantas veces se va a ejecutar. Intentaremos usar un bucle **while** en una estructura como

```
cont = 0
while condicion1:
    ....
    if condicion2:
        ....
        cont += 1
    <actualizamos parametros de condicion1>
return cont
```

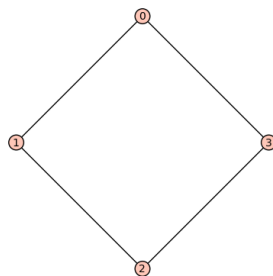
La actualización de los parámetros de los que depende la **condicion1** es lo que debe acercarnos al momento en que al no cumplirse la condición el bucle **while** se para, mientras que la **condicion2** expresa la definición de los objetos que estamos contando.

4.3. Recursión

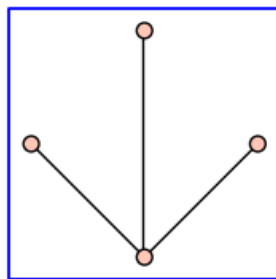
1. La recursión es un procedimiento similar, en muchos aspectos, a la inducción matemática.
2. Decimos que un programa es **recursivo** cuando dentro del bloque de instrucciones del programa hay una llamada al mismo programa con otros argumentos, en general más pequeños.
3. En la definición **recursiva** de una función debe estar previsto un caso inicial que PUEDA PARAR la recursión y tiene que ocurrir que LA RECURSIÓN EFECTIVAMENTE LLEGUE AL CASO INICIAL.
4. Los programas recursivos suelen utilizar una cantidad grande de RAM, ya que, por ejemplo, para calcular una función de n necesitan guardar en memoria la relación entre el valor para n y el valor para $n - 1$, la relación entre el valor para $n - 1$ y el valor para $n - 2$, etc. y no calculan nada hasta que llegan hasta el caso inicial, y entonces empiezan a sustituir para valores crecientes de n .
5. Necesitamos ahora algunas definiciones básicas en la teoría de grafos:
 - a) Un *grafo* consiste en un conjunto de *vértices* V y un conjunto de *aristas* $E \subset V(2)$, con $V(2)$ que denota el conjunto de todos los subconjuntos de V que tienen dos elementos. Representamos geoméricamente un grafo pintando los vértices como puntos y las aristas como líneas que unen vértices.



- b) Un *ciclo* en un grafo es una sucesión ordenada de vértices $v_0, v_1, v_2, \dots, v_n$ tales que $v_n = v_0$ y cada vértice v_i , con $i < n$, está unido por una arista a v_{i+1} .



- c) Un *árbol* es un grafo sin ciclos. En ocasiones elegimos un vértice al que llamamos *raíz* y representamos el árbol con la raíz como el punto más bajo y las aristas “hacia arriba” sin cortarse.



- d) Llamamos *hojas* de un árbol con raíz a los vértices, distintos de la raíz, a los que sólo llega un eje.
- e) Llamamos *profundidad* de un árbol con raíz al máximo del número de aristas entre la raíz y una hoja.
- f) La profundidad y el número de hojas son una medida de lo *frondoso* (=complejo) que es un árbol y por eso nos serán útiles. Dependen de la elección del vértice raíz, pero en nuestro caso tendremos una elección *natural* de raíz.
6. La ejecución de un programa recursivo tiene asociado un *árbol de llamadas* recursivas, con raíz natural la llamada que arranca el programa y vértices cada una de las llamadas a sí mismo, cada una con los valores de los parámetros con los que se llama. Las hojas son todas las llamadas a los casos iniciales de la recursión.

7. EJEMPLOS:

a) FIBONACCI RECURSIVAMENTE:

```
def fibon2(m):  
    if m in [0,1]:  
        return m  
    else:  
        return fibon2(m-1)+fibon2(m-2)
```

Si comparamos el tiempo, usando `time` al comienzo de la línea en la celda de cálculo, que tarda el programa recursivo con el que tarda el iterativo vemos que el iterativo es muchísimo más eficiente.

```
def fibon(m):  
    a,b=1,0  
    for j in xrange(m):  
        a,b=b,a+b  
    return b
```

La única ventaja del recursivo es que el programa es prácticamente igual a la definición de la sucesión de Fibonacci.
¿Qué aspecto tiene el árbol de este programa?

b) FACTORIAL:

```
def fact(m):  
    if m == 0:  
        return 1  
    else:  
        return factorial(m-1)*m
```

¿Qué aspecto tiene el árbol de este programa? ¿Puedes explicar por qué `fibon` recursivo es muy ineficiente mientras que `fact` recursivo es prácticamente tan eficiente como el iterativo?

8. Entonces, los programas recursivos pueden ser muy poco eficientes, aunque no siempre, comparados con un programa similar iterativo, pero tienen la ventaja de que suelen ser más cortos, y (quizá) más fáciles de entender.

9. En ocasiones podemos escribir programas recursivos muy eficientes. Por ejemplo, supongamos un programa que debe manipular una lista de enteros, digamos ordenarla. Un enfoque posible consiste en dividir la lista por la mitad y ordenar las dos mitades (llamada recursiva), y, por último debemos intercalar los elementos de la segunda lista entre los de la primera para obtener la lista original ordenada.

Esto es factible y muy eficiente, cada vez que se incrementa la profundidad en el árbol de llamadas en una unidad el tamaño de la lista se divide por dos. Podemos ver un ejemplo de esta forma de resolver el problema en la hoja [43-PROGR-ordenacion-listas.ipynb](#), y en mayor detalle en la sección [4.5.2](#).

10. Todo programa recursivo se puede traducir en un programa iterativo y viceversa, pero en ocasiones la traducción es de lectura difícil y el resultado puede ser ineficiente. En este curso no queremos entrar en esas profundidades, que, propiamente, pertenecen a la teoría de la computación.
11. Se pueden ver varios ejemplos más de recursiones en la hoja de Sage [42-PROGR-recursiones.ipynb](#).

4.4. Depurando código

1. En primer lugar un código puede tener *errores sintácticos*, es decir, puede ocurrir que el intérprete de Python no lo entienda, no sepa que hacer con él, y produzca mensajes de error:
 - a) Los mensajes de error son, en ocasiones, bastante crípticos pero, al menos, suelen mostrar mediante un angulito debajo de una zona del código el lugar aproximado en el que se ha detectado el error.
 - b) Los errores más simples, y fáciles de arreglar, son los errores de sangrado (“Indentation Error”, “error de alineamiento del código”).
 - c) También es fácil detectar errores como la falta de *los dos puntos* al final de líneas que deben llevarlos (**def**, **for**, **if**, **while**), paréntesis o corchetes no balanceados (i.e., mismo número de abiertos y cerrados en una expresión), errores en el nombre o el número de argumentos de una función, etc.
 - d) En la hoja de Sage [44-PROGR-mensajes-error.ipynb](#) pueden verse algunos ejemplos de errores sintácticos típicos y de los mensajes de error que generan.
2. Una vez que es posible ejecutar, sin errores, el código todavía cabe la posibilidad de *errores semánticos*, es decir, el código produce resultados erróneos:

- a) Conviene usar NOMBRES DIFERENTES para todos las variables que deben ser distintas en un cálculo, y lo mismo se debe aplicar dentro de una hoja de Sage.
- b) Ocurre frecuentemente que podemos ver si los resultados son correctos directamente: si la función ordena listas el resultado debe estar ordenado, si estamos calculando π el resultado debe comenzar como 3.14..., etc.
Siempre hay que tratar de ver si los resultados obtenidos son razonables comparando con lo que esperaríamos *a priori*.
- c) En muchos casos podemos detectar errores semánticos calculando *a mano*, para valores pequeños de sus argumentos, el valor o valores devueltos por la función.
En ocasiones estamos programando una función que ya existe en Sage, y lo más simple es comparar nuestros resultados con los de Sage.
- d) Si encontramos errores semánticos puede deberse, y suele ser el caso, a que no hemos entendido bien el algoritmo que estamos tratando de implementar. En ese caso es muy conveniente ejecutar *en papel*, y para valores pequeños de sus argumentos, nuestro programa incorrecto y comparar con la ejecución, también *en papel*, del algoritmo.
- e) Una de las técnicas más útiles para corregir errores semánticos consiste en mostrar el valor de algunas de las variables al paso del programa por algún punto en su ejecución. Habitualmente hacemos esto intercalando una línea como

```
print var_1,var_2,...,var_n
```

en el lugar adecuado del código.
- f) Un programa puede tener un sangrado sintácticamente correcto pero semánticamente incorrecto: el sangrado indica el alcance del bloque sobre el que se itera o el de las instrucciones que se ejecutan sólo si se cumple la condición, y es posible que ese alcance de un bloque sea incorrecto.
También puede ser incorrecto el orden de las líneas dentro de un bloque (ver página 94).
- g) En criptografía, capítulo 8, manejamos dos funciones: la primera pasa de un texto legible al correspondiente texto encriptado y la segunda debe recuperar el texto legible a partir del encriptado. Cuando programamos estas dos funciones, para un método criptográfico cualquiera, debemos siempre comprobar que se cumple esta condición.
- h) En ocasiones el número de soluciones es conocido *a priori*, de forma que si nuestro programa las calcula es fácil comprobar si las ha encontrado todas. Por ejemplo, supongamos que queremos determinar todas las funciones biyectivas de un conjunto finito con n elementos en sí mismo: sabemos que el número de tales biyecciones es $n!$ de forma que es fácil comprobar, incluso para n grande, si están todas.

- i) Es muy conveniente, si es posible, dividir el código de una función entre varias subfunciones que son llamadas desde la principal (programación estructurada). Si hacemos esto es mucho más fácil encontrar y corregir errores semánticos ya que podemos trabajar de manera independiente con cada una de las subfunciones.
- 3. Por último, un programa puede ser sintácticamente y semánticamente correcto pero muy ineficiente, es decir, puede tardar mucho más tiempo y usar mucha más memoria RAM de la estrictamente necesaria. De cómo mejorar esta situación se tratará en el siguiente capítulo.

4.5. Ejemplos

4.5.1. Órbitas

A lo largo del curso nos vamos a encontrar, en más de una ocasión, con esta situación:

Tenemos una función $f : X \rightarrow X$ de un conjunto X en sí mismo y un valor inicial $x_0 \in X$ y queremos estudiar la sucesión de iterados de x_0 mediante la función f

$$o(x_0) := \{x_0, f(x_0), f^2(x_0), f^3(x_0), \dots, f^n(x_0), \dots\} \subset X$$

La notación $f^n(x_0)$ indica que hemos aplicado sucesivamente n veces la función f a x_0 , es decir,

$$f^n(x_0) := f(f(f(\dots)))(x_0),$$

y la función f aparece n veces.

Cuando queremos pensar geométricamente llamamos al conjunto $o(x_0)$ de todos los iterados de x_0 la **órbita** de x_0 mediante f . Otras veces pensamos de “manera física” y entonces el exponente n es el tiempo variando de forma discreta, es decir a saltos, la función f representa la evolución temporal de un proceso físico, mientras que los elementos del conjunto X son los “estados posibles” del sistema que estamos estudiando.

Por poner un ejemplo concreto, un punto (x, v) en $X = \mathbb{R} \times \mathbb{R}$ podría representar la posición y velocidad de una partícula que se mueve en una única dimensión espacial, y $f(x, v) =: (x_1, v_1)$ sería el punto representando la posición y velocidad de la partícula, calculada de acuerdo a las leyes de Newton, después de un segundo. Para cada estado inicial (x_0, v_0) los iterados de la función f son instantáneas, tomadas de segundo en segundo, del estado de movimiento de la partícula.

Cuando trabajamos con el ordenador las órbitas deben ser finitas, lo que está asegurado si el conjunto X es finito. En Si X es finito aparece necesariamente un punto $x_{i_0} = f^{i_0}(x_0)$ de la órbita de x_0 al que se vuelve cuando seguimos iterando, es decir, tal que $x_{i_0} = f^{i_0}(x_0) = f^{i_1}(x_0) = x_{i_1}$. Decimos que se ha producido un **ciclo**, y cuando X es finito todas las órbitas terminan en un ciclo, que puede consistir en un único punto fijo por f ($f(x) = x$).

Observa que el ciclo no contiene necesariamente al punto inicial x_0 , sin embargo

Ejercicio 1. Demuestra que si X es finito y f es biyectiva, el conjunto X es la unión disjunta de ciclos de f .

Frecuentemente queremos calcular la órbita de un elemento $x_0 \in X$ o bien su longitud, i.e. el número de sus elementos distintos, o incluso los ciclos de una cierta función f .

Entonces vamos a programar una función de SAGE genérica que luego podremos aplicar en casos particulares.

1. Debemos definir la función f que vamos a iterar:

```
def f(?):
    return -----
```

Hay que precisar el argumento (argumentos) de la función y los valores que asignamos a cada elección de los argumentos.

2. La función que calcula la órbita tendrá un bucle **while**, ya que no sabemos *cuanto tenemos que iterar* hasta volver a un punto por el que ya hemos pasado.

```
def orbita(ini,f):
    L = []
    while not(ini in L):
        L.append(ini)
        ini = f(ini) #Actualiza el valor de ini
    return L
```

3. Esta función nos devuelve una lista con primer elemento **ini**, el x_0 de la órbita, y último elemento x_n con la propiedad de que $f(x_n)$ ya está en la lista y eso no ocurre para un n más pequeño.

Si sólo nos interesa la longitud de la órbita, ¿cómo hay que modificar la función?

4. Una generalización del programa anterior trataría de parar el bucle **while** no cuando se cerrara un ciclo sino cuando dejara de cumplirse una condición booleana cualquiera. En primer lugar, deberíamos definir una función

```
def condparada(??):
    if -----:
        return True
    else:
        return False
```

con argumento (argumentos) y condición para que devuelva **True** adecuados a nuestro problema. Usando ésta, y modificando adecuadamente **orbita**, podemos definir otra, por ejemplo **iterahasta(ini,f,condparada)**, que devuelva la lista de todos los valores que se obtienen iterando f a partir de **ini** hasta que se cumple la condición de parada.

5. ¿Qué pasa si cambiamos, en el programa del apartado 2) el orden de las dos líneas a continuación del **while**? ¿Por qué?

4.5.1.1. Collatz

Comenzamos definiendo una función $f(n)$ de un entero positivo n mediante

$$\begin{cases} n/2 & \text{si } n \text{ es par} \\ 3 \cdot n + 1 & \text{si } n \text{ es impar} \end{cases}$$

El problema de Collatz consiste en probar que para todo entero positivo n la órbita de n contiene al 1. Como el 1 produce un ciclo

$$1, f(1) = 4, f(4) = 2, f(2) = 1,$$

si fuera verdad querría decir que todas las órbitas “mueren” en ese ciclo y no antes.

A pesar de que tiene una apariencia bastante inocente, NO SE SABE si la respuesta al problema de Collatz es afirmativa o negativa.

Puedes ver un par de artículos sobre el problema de Collatz [en este enlace](#) y [en este otro](#). El segundo es más elemental, y, por tanto, un poco más asequible. En cualquier caso se trata de un problema difícil que se ha intentado atacar con una gran variedad de métodos y, hasta ahora, sin resultado.

Ejercicio 2.

1. Escribe un programa para comprobar si el problema tiene solución afirmativa para los enteros positivos n menores que un entero n_0 dado.
2. Escribe un programa que, dado un entero n_0 , produzca una lista que tenga en la posición n , $n \leq n_0$, el mínimo entero N tal que $f^N(n) = 1$.
3. ¿Por qué crees que el problema de Collatz es tan difícil?

4.5.2. Ordenación

Ordenar una lista de enteros desordenados es una de las operaciones básicas que se pueden hacer con un ordenador, y, cuando la lista es enorme, puede exigir demasiado tiempo de cálculo. En este ejercicio hay que programar tres de los métodos existentes para ordenar listas, vamos a ordenarlas en orden creciente, y compararemos los resultados

1. PRIMER ALGORITMO (INSERTION SORT):

La forma más sencilla, pero no la más eficiente, de ordenar una lista consiste en comparar cada par de elementos e invertirlos si el mayor aparece antes en la lista.

Ejercicio 3.

- a) Define una función de Sage `intercambiar(L,i,j)`, con argumentos una lista y un par de enteros, y tal que devuelva la lista que se obtiene intercambiando el elemento i -ésimo de la lista con el j -ésimo si $i < j$ pero el i -ésimo elemento de la lista es mayor que el j -ésimo.
- b) Define una segunda función `ordenar1(L)` que, usando la del apartado anterior, devuelva la lista L ordenada. Observa que para que la lista quede ordenada hay que comparar cada elemento de la lista con TODOS los siguientes, es decir, necesitamos un bucle doble para recorrer los pares (i, j) con $i < j$.

2. SEGUNDO ALGORITMO (MERGESORT):

Ejercicio 4.

- a) Este es un algoritmo recursivo que también utiliza una función auxiliar.

Define una función de Sage `intercalar(L,L1,L2)` de tres listas que haga lo siguiente: si el primer elemento de `L1` es menor o igual que el primero de `L2` debe quitar el primer elemento de `L1`, añadirlo a `L` y volver a llamar a la función `intercalar(L,L1,L2)`; en caso contrario debe quitar el primer elemento de `L2`, añadirlo a `L` y volver a llamar a la función `intercalar(L,L1,L2)`. Las recursiones deben parar cuando una de las dos listas (`L1` o `L2`) sea vacía y devolver la suma de las tres listas `L+L1+L2`.

- b) Define una función de Sage `ordenar2(L)` que en los casos `len(L)=0` o `len(L)=1` devuelva la lista `L`, y devuelva `intercalar([],L1,L2)` con `L1` y `L2` las listas que se obtienen aplicando `ordenar2` a las dos “mitades” de `L` (si el número de elementos de `L` es impar una de las “mitades” tiene un elemento más que la otra). Es importante observar que cuando llamamos a `intercalar` las listas `L1` y `L2` ya están ordenadas.

- c) Compara la eficiencia de los dos métodos anteriores generando una lista de 800 enteros aleatorios comprendidos entre -1000 y 1000

```
L = [randint(-1000,1000) for muda in xrange(800)]
```

y midiendo el tiempo que el ordenador tarda en ordenarlas

```
time ordenar1(L); time ordenar2(L)
```

3. TERCER ALGORITMO (QUICKSORT):

- a) Si la longitud de la lista L es menor o igual que 1 la devolvemos.
- b) Si la longitud en $n > 1$ creamos tres listas: $L1$ contiene todos los elementos de L que son menores que $L[0]$, $L2$ contiene todos los elementos de L que son mayores que $L[0]$, y $L3$ contiene todos los elementos de L que son iguales a $L[0]$.
- c) Aplicamos recursivamente el procedimiento a $L1$ y $L2$, para obtener $M1$ y $M2$, y devolvemos $M1 + L3 + M2$.

Ejercicio 5.

- a) Define una función `ordenar(L)` que devuelva la lista L ordenada y utilice el algoritmo propuesto. Comprueba que realmente ordena una lista de longitud 20 formada por enteros aleatorios del intervalo $[-100, 100]$.

- b) Sage dispone del método `L.sort()` para ordenar listas. Genera una lista de 10^5 números aleatorios del intervalo $[-10^6, 10^6]$ y compara los resultados y tiempos usando tu función y la de Sage.
- c) Una variante de este algoritmo consiste en utilizar en el paso 2 la comparación con un elemento aleatorio de la lista L en lugar de comparar con el primero. Implementa esta variante y estudia si es más eficiente o no que el original.
- d) Queremos encontrar la menor diferencia entre dos enteros cualesquiera pertenecientes a una lista L de enteros. Una idea consiste en ordenar la lista y luego buscar la menor diferencia entre enteros consecutivos de la lista ordenada. Define una función que haga esto.

4. CUARTO ALGORITMO (BUCKET SORT):

- a) La función que vamos a definir tiene el formato `ordenar(L, i, j)` con L la lista a ordenar e i, j enteros del intervalo $[0, \text{len}(L) - 1]$.
- b) Comparamos el elemento $L[i]$ con $L[j]$, y si este último es menor los intercambiamos.
- c) Si $j - i + 1 > 2$ aplicamos la función (`ordenar`) a L con $j - (j - i + 1)/3$ en lugar de j y sin cambiar el valor de i , luego con $i + (j - i + 1)/3$ en lugar de i y sin cambiar el valor de j , y finalmente, otra vez con $j - (j - i + 1)/3$ en lugar de j sin cambiar el valor de i .
- d) Devolvemos L .

Ejercicio 6.

- a) Define la función `ordenar(L, i, j)` propuesta. Cuando se invoca en la forma `ordenar(L, 0, n)`, con $n = \text{len}(L) - 1$ debe devolver la lista L ordenada.
- b) Comprueba que realmente ordena una lista de longitud 20 formada por enteros aleatorios del intervalo $[-100, 100]$.
- c) Sage dispone del método `L.sort()` para ordenar listas. Genera una lista de 10^3 números aleatorios del intervalo $[-10^6, 10^6]$ y compara los resultados y tiempos usando tu función y la de Sage.
- d) En el apartado anterior obtendrás tiempos grandes, del orden de un minuto, y queremos estudiar el motivo. Modifica tu función para incluir un contador que nos informe del número de veces que se ha producido el intercambio del punto b en la descripción del algoritmo.

En esta [página de la Wikipedia](#) puedes encontrar descripciones de diversos algoritmos para ordenar listas, y ES UN BUEN EJERCICIO PROGRAMAR algunos y comparar resultados.

4.6. Ejercicios

Cuando en los ejercicios que siguen se utilice un bucle **for** hay que intentar usar las dos modalidades mostradas en las páginas [77](#) y [81](#).

1. Mejorar los programas `gen_subcadena` para que la salida $C1$ contenga información acerca de la posición que ocupa cada dígito de los que añadimos a $C1$.
2. Un par de enteros primos se dicen “gemelos” si difieren en 2 unidades. Definir una función de n que devuelve una lista de todos los pares de primos gemelos menores que n .
3. Define una función, de dos enteros k y a , que cuente el número de primos gemelos en cada uno de los subintervalos $[kt, k(t+1)]$ de longitud k dentro del intervalo $[0, ka]$.
4. Define una función que cuente el número de enteros primos menores que N de la forma $n^2 + 1$ para algún n .
5. Dado un entero N define una función de N que devuelva el subintervalo $[a, b]$ de $[1, N]$ más largo tal que no contenga ningún número primo.
6. Dado un entero, que escribimos en la forma $N = 10 \cdot x + y$ con y la cifra de las unidades, definimos $F(N) := x - 2 \cdot y$.
 - a) Demuestra que N es múltiplo de 7 si y sólo si $F(N)$ lo es.
 - b) Estudia las órbitas de F en el conjunto de los enteros positivos, tratando de enunciar un criterio de divisibilidad de N entre 7 en términos de la órbita de N mediante F .
7. Consideramos la función $F : \mathbb{N}^* \rightarrow \mathbb{N}^*$ que a cada entero positivo le asocia la suma de los cuadrados de sus dígitos.
 - a) Estudia las órbitas de F , tratando de determinar el *comportamiento a largo plazo* de las iteraciones de F .
 - b) Después de obtener tus conclusiones acerca de las órbitas, demuéstalas formalmente. Para eso es importante entender cómo decrece un entero N , al aplicarle F , dependiendo de su número de dígitos.

8. Queremos estudiar las órbitas de la función $f(n) := n^2$ en el conjunto finito de clases de restos módulo m . En particular, queremos determinar las órbitas cerradas (también llamadas *ciclos*), es decir, órbitas de elementos n tales que aplicando repetidas veces f volvemos a obtener el mismo n . Puede haber ciclos que consistan en un único elemento n tal que $f(n) = n$, y decimos en este caso que n es un punto fijo de f .
- a) Define una función de Sage $\text{ciclos}(f, m)$ que devuelva una lista de todos los ciclos de f en el conjunto de clases de restos módulo m , cada ciclo una lista de los enteros que forman parte del ciclo. En principio, no hay problema en que haya ciclos repetidos en la lista que devuelve tu función.
 - b) Experimenta con los resultados de tu función, para diversos valores de m primos y compuestos, y trata de obtener conjeturas razonables sobre los ciclos de f .

Al resolver este ejercicio conviene tener en cuenta las definiciones y resultados mencionados en las dos primeras secciones del capítulo siguiente, y, en particular, todo lo que se refiere a la **noción de orden**.

9. Encontrar la condición necesaria y suficiente sobre el entero n para que $1 + 2 + 3 + \dots + n$ divida exactamente a $n!$. Una vez encontrada hay que intentar demostrar la equivalencia.
10. Encontrar la condición necesaria y suficiente para que existan infinitos múltiplos de un entero n que se escriban únicamente con unos en base 10. Una vez encontrada hay que intentar demostrar la equivalencia. ¿Ocurrirá algo similar si queremos que infinitos múltiplos se escriban usando únicamente otro dígito (2, 3, etc.)?
11. La cifra dominante de un entero es la primera por la izquierda que no es nula. Encontrar los dígitos que pueden ser la cifra dominante al mismo tiempo de 2^n y 5^n , con el mismo exponente n . Una vez encontrados, hay que intentar demostrar que esos dígitos son los únicos posibles.
12. Demuestra que para $n > 1$ el entero $n^4 + 4$ es siempre compuesto.
13. En el [sitio web projecteuler](#) pueden encontrarse los enunciados de casi 500 problemas de programación. Dándose de alta en el sitio es posible, pero no es lo recomendable, ver las soluciones propuestas para cada uno de ellos.

Tal como están enunciados muchos de ellos son difíciles, ya que no se trata únicamente de escribir código que, en principio, calcule lo que se pide, sino que debe ejecutar el código en un ordenador normal de sobremesa, en un tiempo razonable, para valores de los parámetros muy altos. Es decir, lo que piden esos ejercicios es que se resuelva el problema mediante

un código correcto y muy eficiente. Sin embargo, en primera aproximación, podemos intentar producir código sintáctico y semánticamente correcto, que no es poco, y dejar el asunto de la eficiencia para más adelante.

Muchos de estos problemas de programación tienen una base matemática fuerte, y eso es lo que sobre todo nos interesa.

- a) Goldbach conjeturó que todo entero compuesto e impar es la suma de un primo y el doble de un cuadrado. Así, por ejemplo, $9 = 7 + 2 \cdot 1^2$, $15 = 7 + 2 \cdot 2^2$, $21 = 3 + 2 \cdot 3^2$, etc. Esta conjetura resultó ser falsa. Determina el menor entero que no cumple lo conjeturado por Goldbach.
- b) Existen enteros, por ejemplo 145, que son iguales a la suma de los factoriales de sus dígitos. Determina todos los enteros con esta propiedad.
- c) Determina todas las tripletas de enteros primos de 4 cifras tales que cumplen las dos condiciones siguientes:
 - 1) Los 3 enteros están en progresión aritmética, es decir, el segundo menos el primero es igual al tercero menos el segundo.
 - 2) Los tres enteros de la triplete tienen las mismas cifras y cada cifra aparece el mismo número de veces en cada uno de ellos.

Por ejemplo, (1487, 4817, 8147) es una de las soluciones.

- d) Sea (a, b, c) una triplete de enteros positivos tal que existe un triángulo rectángulo con la longitud de los lados igual a los enteros de la triplete. Podemos decir, por ejemplo, que una tal triplete es rectangular. Sea p el perímetro de un tal triángulo. Para $p \leq 1000$ determina el perímetro p_m para el que existe el mayor número de tripletas rectangulares distintas con ese perímetro.
- e) Un **primo de Mersenne** es un entero primo de la forma $2^p - 1$ con p primo. No es difícil ver que si $2^n - 1$ es primo, entonces el exponente n debe ser también primo, pero el recíproco es falso. Los primos de Mersenne son los mayores conocidos porque hay criterios de primalidad bastante eficientes para candidatos a ser primo de Mersenne. En 2004 se descubrió un primo muy grande que no es de Mersenne, concretamente se trata de $P := 28433 \times 2^{7830457} + 1$.
 - 1) Determina el número de cifras decimales de P .
 - 2) Determina las últimas, por la derecha, diez cifras decimales de P .
 - 3) Determina las primeras, por la izquierda, diez cifras decimales de P .

Se entiende que, aunque fuera posible calcular completamente P , debe hacerse este ejercicio sin pasar por ese cálculo.

f) Hay polinomios de grado 2 que, como $p(x) := x^2 + x + 41$, toman valores primos para los primeros valores enteros consecutivos de x . El polinomio $p(x)$ indicado toma valores primos para $x = 0, 1, 2, \dots, 39$, pero el valor es compuesto para $x = 40$. Entonces, para $p(x)$ se obtiene una sucesión de 40 primos al evaluarlo en los enteros consecutivos del intervalo $[0, 39]$.

Determina, de entre todos los polinomios de la forma $x^2 + ax + b$, con a y b de valor absoluto menor o igual a 1000, el polinomio que produce el mayor número de valores primos al evaluarlo en enteros consecutivos $x = 0, 1, 2, \dots$. Indica también el número de valores primos obtenido, que será mayor o igual a 40.

g) Otros ejercicios del proyecto Euler que podemos recomendar son los números

2, 4, 8, 10, 12, 21, 24, 23, 26, 28, 29, 30, 35, 36, 37, 41, 45, 50, 57.

Son ejercicios del comienzo de la lista y, en general, son m'as sencillos que los del final. Otros pocos, situados alrededor del número 400 serían

h) EJERCICIO #401: Pide que se evalúe la suma de todos los cuadrados de los divisores de un número n , y luego que se sumen esas sumas para n entre 1 y N . Puede ser difícil evaluar esas sumas eficientemente para N muy grande.

i) EJERCICIO #413: Es un ejercicio en el manejo de cadenas de caracteres.

j) EJERCICIO #414: En este se utilizan listas y el cálculo de **órbitas**.

k) EJERCICIO #421: Factorización de enteros y de polinomios. Sage será de gran ayuda para resolverlo.

l) EJERCICIO #429: Otro ejercicio sobre divisores de un entero.

14. Algunos programas recursivos son muy parecidos a una demostración *por inducción*. Consideremos, por ejemplo, el primer ejercicio en **la lista de ejercicios del capítulo 2**:

Se trata de comprobar, por inducción, que

$$1^2 + 2^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}. \quad (4.1)$$

En la demostración por inducción debemos ver que:

a) *Caso inicial*: $1^2 = \frac{1 \cdot 2 \cdot 3}{6}$, que es claro.

b) Que si suponemos, *hipótesis inductiva*, que

$$1^2 + 2^2 + \dots + (n-1)^2 = \frac{(n-1)n(2n-1)}{6},$$

entonces también debe ser cierta (4.1).

DEFINE RECURSIVAMENTE una función de Sage, por ejemplo **def** comprueba(**N**):..., que sirva para asegurarnos de que la fórmula (4.1) es cierta desde $n = 1$ hasta $n = N$,

- a) Debemos usar el *caso inicial* como condición de parada de la recursión. Entonces, debemos tener un bloque **if** **N** == 1: ...
- b) En lugar de la *hipótesis de inducción* tenemos que incluir en algún lugar del programa una llamada recursiva a la propia función comprueba(**N**-1).

ESCRIBE PROGRAMAS RECURSIVOS para los ejercicios de demostración por inducción de la lista mencionada.

Apéndice C

Máquina virtual

En computación una *máquina virtual* es un programa que se ejecuta en otra máquina, virtual o física, y que simula el funcionamiento de un ordenador. Al proceso de ejecutar aplicaciones dentro de máquinas virtuales se le llama *virtualización* de aplicaciones, y ciertamente es una tecnología que está actualmente bastante de moda.

Por ejemplo, en las máquinas de nuestro Laboratorio no está instalado *MS Windows*®, pero como hay algunos cursos en los que se usa es necesario que, al menos, se pueda ejecutar en una máquina virtual. En nuestro caso usamos el programa *VirtualBox* que nos permite ejecutar *MS Windows*®, y sus programas, dentro de Linux¹.

En esta sección vamos a estudiar el funcionamiento de la máquina virtual ² más simple que existe, que aún así es capaz de ejecutar cualquier algoritmo finito.

Comenzamos describiendo la máquina:

1. Un ordenador necesita *memoria RAM* para almacenar los datos sobre los que trabaja. Normalmente, también tiene un *disco duro (HD)* o *disco de estado sólido (SSD)* en el que almacenar permanentemente esos datos.

¹Aunque pueda parecer increíble, es posible ejecutar software de *Windows*® dentro de Linux, usando por ejemplo *Wine*, de manera bastante más rápida que directamente en *Windows*®. Puedes leer una breve explicación en los párrafos 2 y 3 de esta [página](#).

²Técnicamente, esta máquina es un ejemplo de *máquina de registros* conocida como “modelo ábaco” de Lambek.

Nuestra máquina virtual va a almacenar sus datos en una lista de enteros no negativos de longitud arbitraria. Esta lista, a la que llamaremos **datos**, va a simular al mismo tiempo la memoria RAM y el disco duro.

2. Los ordenadores tienen *procesadores (CPU)* en los que se ejecutan las *instrucciones* contenidas en los *programas (software)*. Los datos que el procesador lee de la memoria RAM o del disco son modificados, procesados, y devueltos a la memoria o al disco.

Nuestros programas serán *listas de instrucciones* elementales y describiremos el procesador de nuestra máquina virtual detallando las instrucciones que admite:

- a) El primer tipo de instrucciones, lo llamaremos 'A', va a sumar una unidad a un cierto elemento de la lista **datos** y a continuación va a saltar a otra instrucción del programa que estamos ejecutando.

Una instrucción de tipo 'A' consistirá en una lista de longitud 3 de la forma ['A',i,j], con *i* un entero no negativo que determina un elemento, **datos**[*i*] de la lista **datos** y *j* otro entero no negativo que nos indica la instrucción del programa que se va a ejecutar a continuación.

- b) Una instrucción de tipo 'S' va a restar una unidad a un cierto elemento de la lista **datos** y a continuación va a saltar a otra instrucción del programa que estamos ejecutando.

Sin embargo, y esto es MUY IMPORTANTE, como los elementos de la lista **datos** no pueden ser negativos, cuando nos encontremos que el elemento al que habría que restarle una unidad ya es cero no restamos y saltamos a otra instrucción directamente.

En consecuencia, las instrucciones de tipo 'S' son listas de longitud 4 de la forma ['S',i,j,k] con *i*, *j* y *k* enteros no negativos: *i* un índice de la lista **datos**, *j* la instrucción a la que saltamos si **datos**[*i*] $\neq 0$ y *k* la instrucción a la que saltamos si **datos**[*i*]=0.

- c) Finalmente, una instrucción de tipo 'E' es una instrucción de fin del programa y consiste de una lista de longitud 1 que únicamente contiene el carácter 'E'. Cuando el procesador se encuentra una instrucción de tipo 'E' para la ejecución del programa.

3. Por último, los ordenadores necesitan un *sistema operativo (OS)* que gestione la asignación de recursos físicos para la ejecución de los programas.

En nuestra máquina virtual el sistema operativo consistirá en una función que tendrá como argumentos un programa y una lista de datos, modificará la lista de datos ejecutando en sucesión las instrucciones del programa y devolverá el estado final de la lista de datos.

Implementamos esta máquina virtual como un programa en Python, con dos funciones una que describe el procesador y la otra el sistema operativo:

```
def procesador(instruccion,datos,end):
    if instruccion[0] == 'A':
        datos[instruccion[1]] += 1
        estado = instruccion[2]
    elif instruccion[0] == 'S':
        if datos[instruccion[1]] != 0:
            datos[instruccion[1]] -= 1
            estado = instruccion[2]
        else:
            estado = instruccion[3]
    elif instruccion[0] == 'E':
        estado = end
    return datos,estado

def sistema_op(programa,datos):
    estado = 0
    end = len(programa)
    while estado != end:
        datos,estado = procesador(programa[estado],datos,end)
    return datos
```

Ya podemos comenzar a programar nuestra máquina:

```
programa1=[[ 'S',0,1,2],[ 'A',1,0],[ 'E']]
programa2=[[ 'S',0,1,3],[ 'A',1,2],[ 'A',2,0],[ 'S',2,4,5],[ 'A',0,3],[ 'E']]
```

¿Qué hacen estos dos programas? Lo primero que debemos determinar, para cada uno de ellos, es la longitud de la lista **datos** que procesan. Para el primer programa vemos que el máximo del segundo elemento en cada instrucción es 1 luego la lista de datos debe ser de longitud dos (puede ser de longitud mayor pero entonces el programa no modificará los elementos a partir del tercero), mientras que para el segundo vemos que la longitud de la lista de datos es tres.

A continuación, ejecuta en papel cada uno de los dos programas, usando enteros pequeños como elementos de la lista de datos, y trata de entender su funcionamiento y uso.

COMENTARIOS:

1. Nuestra máquina virtual tiene un conjunto de instrucciones minimal, lo que hace que los programas sean simples pero muy largos. Hablando muy en general, nuestra máquina virtual tiene un procesador de tipo RISC (Reduced Instruction Set Computer) mientras que los ordenadores personales tienen casi todos procesadores CISC (Complex Instruction Set Computer).

Un programa que se va a ejecutar en un procesador RISC consistirá en muchas más instrucciones elementales que uno para CISC, pero estas instrucciones son mucho más sencillas y se pueden ejecutar más rápido. Típicamente, los procesadores RISC ejecutan más instrucciones por ciclo de reloj que los CISC lo que se compensa con tener que ejecutar más instrucciones.

2. Por otra parte, el tipo de programas que utilizamos en nuestra máquina virtual se conoce como de *bajo nivel*, lo que significa que se utilizan directamente las instrucciones que admite el procesador.

En los lenguajes de *alto nivel* que usamos habitualmente para programar es necesario utilizar un *compilador o intérprete* que traduce cada una de las instrucciones de nuestro programa de alto nivel a un montón de instrucciones del procesador.

El lenguaje de programación C es *compilado*, lo que significa que el código pasa por un programa compilador que produce un ejecutable en el que las instrucciones ya son las del procesador. En cambio Python es un lenguaje *interpretado* que usa un programa, llamado intérprete, que va traduciendo *sobre la marcha* las instrucciones del programa a instrucciones del procesador.

Los lenguajes de programación interpretados son mucho más flexibles que los compilados, pero también son más lentos.

3. Debe ser claro que con las instrucciones 'A' y 'S' podemos ejecutar bucles **for** y crear ramificaciones en la ejecución del programa similares a los **if**. Es un poco limitado porque el único caso en que la ejecución del programa se bifurca es en las instrucciones de tipo 'S': si la instrucción va a actuar sobre `datos[i]` hace algo distinto si ese valor es cero o si no lo es.

Disponiendo de **for** e **if** la máquina virtual puede, de hecho, ejecutar cualquier algoritmo finito, aunque los programas, para casi cualquier algoritmo, serían inmensos.

4. Puede parecer que la máquina virtual simula una calculadora y no un ordenador. Sin embargo, puede realizar manipulaciones arbitrarias sobre cadenas de bits y, por tanto, es un ordenador.

EJERCICIOS:

Programar esta máquina virtual es un buen ejercicio cuando se comienza a programar sin experiencia previa. En particular, se ven bastante bien los errores como bucles infinitos, programas sintácticamente correctos pero semánticamente incorrectos, etc.

1. Programa, usando como ayuda los dos ejemplos suministrados, el producto de dos enteros positivos.
2. Programa la resta de dos enteros n y m con $n \geq m$.
3. Programa la suma y resta con enteros positivos o no.
4. Programa la división con resto de enteros positivos.
5. Programa las potencias de exponente entero de enteros positivos.

Apéndice D

Tortuga

El lenguaje de programación LOGO fué desarrollado en el MIT, comenzando en los años 60, como un lenguaje bien adaptado a la enseñanza de los conceptos básicos de la programación de ordenadores. Aunque se trata de un lenguaje bastante completo, la parte que se hizo más popular fué su módulo gráfico, *la tortuga*, que ya desde finales de los 60 se ha utilizado en algunas escuelas para enseñar a programar a niños pequeños.

Originalmente la tortuga era un robot móvil controlado por un programa escrito en LOGO y que podía dibujar en el suelo figuras porque disponía de un bolígrafo. Pronto se pasó de dibujar en el suelo con un robot a dibujar en la pantalla del ordenador.

La programación de gráficas es una buena manera de comenzar a programar porque vemos inmediatamente el resultado del programa y, si no corresponde a lo que queremos, vamos corrigiéndolo y estudiando en cada modificación cómo cambia el resultado. En el caso en que no hayas programado antes, es muy recomendable que dediques un tiempo a practicar con la tortuga.

Python dispone de un módulo, *turtle*, que permite programar la tortuga usándolo en lugar de LOGO, y que funciona desde dentro de Sage. Desgraciadamente, es imposible que funcione en el servidor de Sage y se debe ejecutar siempre en la máquina local, bien en el Laboratorio o en tu propia máquina.

Para que funcione la tortuga en Sage hay que instalar un paquete. En Ubuntu o Debian se hace mediante la instrucción

```
sudo apt-get install tk tk-dev python-tk.
```


Podemos comprobar que funciona correctamente evaluando en una celda de Sage

```
import turtle as tt
tt.forward(1500)
```

La primera línea carga en la hoja el módulo **turtle** de Python, y la segunda debe abrir una nueva ventana en la que hay dibujada una flecha horizontal cuya longitud depende del argumento 1500.

Todas las intrucciones de control de la tortuga deben tener el formato **tt.instruccion(parámetros)**. Con la segunda líneas la tortuga avanza al frente 150 unidades y se para. La flecha es horizontal porque, por defecto, la tortuga está inicialmente *mirando al Este*.

La tortuga tiene un eje de simetría, y un sentido dentro de ese eje, y su orientación viene determinada por el ángulo que el eje forma con la horizontal medido en sentido contrario a las agujas del reloj.

INSTRUCCIONES BÁSICAS DEL PAQUETE **turtle**:

1. Las instrucciones **tt.window_width(x)** y **tt.window_height(y)** fijan el tamaño, en píxeles, de la ventana en la que se va a mover la tortuga. Un pixel es el tamaño de un punto, en realidad un cuadradito, en la pantalla que usemos, y por tanto, depende de ella. En el laboratorio la resolución de las pantallas es *****.

2. MOVIMIENTO:

- a) La instrucción **tt.forward(x)** hace avanzar la tortuga x píxeles al frente, mientras que **tt.back(x)** la hace retroceder.
- b) La instrucción **tt.left(a)** gira el eje de la tortuga a grados en sentido contrario a las agujas del reloj, mientras que **tt.right(a)** lo hace en sentido de las agujas.

Si queremos cambiar a radianes basta ejecutar **tt.radians()**, y todas las medidas de ángulos, a partir de ese momento, serán en radianes.

- c) La velocidad con la que se mueve la tortuga se controla con **tt.speed(arg)**, con **arg** igual a *"fast"*, *"normal"*, *"slow"*, o *"slowest"*.

3. POSICIÓN:

- a) **tt.pos()** devuelve las coordenadas de la posición de la tortuga en el momento en que se ejecuta.

- b) `tt.heading()` devuelve el ángulo que la tortuga, su eje, forma con la horizontal, medido en grados y en el sentido contrario a las agujas del reloj. Si se ha ejecutado en la hoja `tt.radians()` la medida en es radianes. ¿Cómo se volverá a medidas es grados?
- c) `tt.setposition(x,y)` lleva la tortuga al punto de coordenadas (x,y) en línea recta y dibujando salvo que se haya levantado el bolígrafo mediante `tt.penup()`, y no se haya bajado posteriormente.
- d) `tt.setheading(a)` deja el eje de la tortuga formando un ángulo a con la horizontal.

4. BOLÍGRAFO:

- a) La instrucción `tt.penup()` levanta el bolígrafo del papel, deja de dibujar, mientras que `tt.pendown()` lo vuelve a bajar.
- b) `tt.pencolor(color)` sirve para cambiar el color de tinta del bolígrafo. El color puede ser un nombre, como *"red"*, *"blue"*, *"green"*, *etc.* o, más en general, el código hexadecimal de un color RGB. Puedes ver algunos de esos códigos en esta [tabla](#).
- c) `tt.pensize(ancho)` determina el ancho en píxeles del trazo del bolígrafo.

En caso de necesidad, puedes consultar el [conjunto completo de instrucciones](#) del módulo `turtle`.

Es importante entender que el movimiento de la tortuga se determina siempre *localmente* respecto a la posición que ocupa en cada momento. Es como si la tortuga llevara encima un sistema de referencia con respecto al que se interpretan en cada momento las instrucciones que vienen a continuación.

Ejemplos y ejercicios

1. `def poligono(n,R):`
*'''Poligono regular de n lados inscrito
 en una circunferencia de radio R'''*
`tt.radians()`
`A = 2*pi.n()/n`
`L = 2*sin(A/2)*R ##Lado del pol\ '{i}gono`
`tt.penup()`

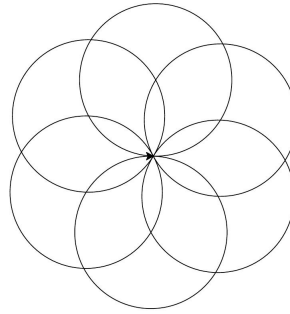
```
tt.forward(R)
tt.left((pi.n()+A)/2)
tt.pendown()
for j in xrange(n):
    tt.forward(L)
tt.left(A)
tt.penup()
tt.home()
```

Si tomamos n grande la *gráfica parece una circunferencia*.

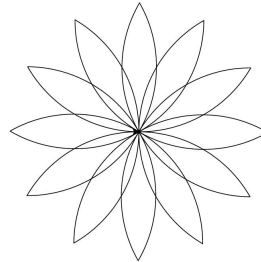
2. Modifica el programa anterior para que, partiendo de la posición en que esté la tortuga, dibuje un arco de α radianes en una circunferencia de radio R .

¿Hace falta dar como dato el centro de la circunferencia? No hace falta si entendemos que la posición original de la tortuga determina la recta tangente, en el punto inicial, a la circunferencia que queremos dibujar. Llamemos L a esa recta, que es el eje de simetría de la propia tortuga en su posición inicial. Entonces el centro de la circunferencia es un punto en la recta perpendicular a L que dista R del punto inicial. Hay dos de esos puntos, uno en cada semiplano de los determinados por L . Entonces hay dos arcos que resuelven el problema, y podemos definir dos programas llamados, por ejemplo, arcoL y arcoR, uno dibuja el arco girando hacia la izquierda y el otro hacia la derecha.

3. Escribe un programa que produzca esta figura:



4. Escribe otro programa que produzca esta otra figura:



5. Escribe un programa que produzca una casa como las que dibujan los niños pequeños, hecha de segmentos y arcos de circunferencia. Intenta producir el mayor número posible de los detalles que suelen incluir esas casas (el árbol, el camino que lleva a la puerta, ¿el humo de la chimenea?, etc.)
6. ¿CONTINUARÁ?

Capítulo 5

Complementos

En este capítulo se recogen pequeñas introducciones a algunos temas, relativos a la programación, más avanzados. Algunos de ellos los usaremos alguna vez a lo largo del curso, y otros no los usaremos pero nos parece conveniente que se sepa de su existencia. Concretamente,

1. Se discuten brevemente los cambios entre bases de los SISTEMAS DE NUMERACIÓN. Puede ser innecesario si se ha visto el tema en el Bachillerato.
2. TRUCOS: en esta sección se recogen algunas ideas para programar que, posiblemente, son reutilizables. Esperamos que pueda ir creciendo a lo largo del curso.
3. FUERZA BRUTA: cuando no se nos ocurre algo mejor siempre podemos intentar un método de *fuerza bruta*. En problemas finitos, o que podamos aproximar mediante un problema finito, será posible este tipo de tratamiento, aunque seguramente muy poco eficiente.
4. CÁLCULO EN PARALELO: Sage dispone de un procedimiento básico para ejecutar en paralelo bucles **for** en máquinas con varios núcleos. No es verdadero paralelismo porque no hay *comunicación* entre los núcleos, pero en condiciones favorables nos puede ahorrar tiempo de espera.

5. EFICIENCIA: se discute la forma de medir tiempos de ejecución y uso de la memoria RAM, junto con métodos, Cython y numpy, para mejorar dramáticamente la eficiencia del cálculo numérico en Sage. Hay que tener en cuenta que los tiempos de cálculo que obtenemos al evaluar la eficiencia dependen mucho de la máquina que estemos usando.

Esta sección intenta resumir el contenido de parte del [Bloque II](#) en las notas de Pablo Angulo mencionadas en el Prólogo.

6. ITERADORES: por último, dentro de la subsección sobre la memoria RAM, se mencionan los *iteradores* o *generadores* como métodos para ahorrar RAM en la ejecución de bucles.

5.1. Sistemas de numeración

Elegida una base $b \geq 2$, podemos representar los números enteros como “polinomios” en b con coeficientes los dígitos en base b , es decir, símbolos que representan los enteros entre cero y $b - 1$. Si $b \leq 10$ se utilizan los dígitos decimales habituales, y si $b > 10$ se utilizan a continuación letras. Así por ejemplo, si la base b es 16 los dígitos son

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F,$$

con A representando el dígito en base 16 que corresponde al decimal 10, B el que corresponde al 11, etc.

Un número entero escrito en base 16 puede ser $7C041F$ que corresponde al número en base 10

$$15 + 1 \cdot 16^1 + 4 \cdot 16^2 + 0 \cdot 16^3 + 12 \cdot 16^4 + 7 \cdot 16^5 = 8127519.$$

Es claro que usando una base más grande se consigue una notación mucho más compacta.

Las expresiones que representan un entero en una base dada no son verdaderos polinomios porque la variable ha sido sustituida por la base del sistema de numeración, y los coeficientes no son arbitrarios sino *dígitos* menores que la base.

Sin embargo, las operaciones entre polinomios, suma, multiplicación y división con resto, y entre enteros en una base de numeración b son esencialmente las mismas. Así, por ejemplo, el algoritmo para efectuar la división con resto es prácticamente el mismo para polinomios y para números enteros.

Una explicación más detallada, con algunos ejercicios, de este asunto puede encontrarse en este [documento](#).

5.1.1. Cambios de base

Habitualmente usamos el sistema de numeración decimal, con base $b = 10$, pero podríamos usar cualquier otro, y en computación se usan, sobre todo, los sistemas binario ($b = 2$), octal ($b = 8$) y hexadecimal ($b = 16$).

Los cambios de base de numeración son simples, y los hacemos pasando a través de la base 10:

1. Un entero escrito en base b , $a_n a_{n-1} a_{n-2} \dots a_1 a_0$ se pasa a base 10 evaluando en base 10 su correspondiente polinomio en la variable b

$$a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + a_{n-2} \cdot b^{n-2} + \dots + a_1 \cdot b + a_0.$$

Hemos visto un ejemplo con $b = 16$ al comienzo de esta sección.

2. Al revés, si tenemos un entero N en base 10 y lo queremos pasar a base b , es decir, escribirlo en la forma

$$a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + a_{n-2} \cdot b^{n-2} + \dots + a_1 \cdot b + a_0,$$

debemos, en primer lugar dividir N entre b y el resto es el dígito de las unidades a_0 en base b . Esto se debe a que podemos escribir

$$a_n \cdot b^n + \dots + a_1 \cdot b + a_0 = b \cdot (a_n \cdot b^{n-1} + \dots + a_1) + a_0.$$

Para calcular el segundo dígito a_1 debemos, dividir el primer cociente $a_n \cdot b^{n-1} + \dots + a_1$ entre la base b y el resto es a_1 . Continuamos en la misma forma hasta llegar a un cociente menor que b , y en ese momento el proceso necesariamente para y hemos obtenido la expresión en base b de N .

5.1.2. Sistemas de numeración en Sage

En Sage podemos obtener los dígitos en una base b de un entero decimal D mediante la instrucción `(D).digits(base=b)` que devuelve una lista de los dígitos ordenados según potencias decrecientes de b , es decir, el primer elemento de la lista es el dígito con exponente de b más alto.

Las instrucciones `bin(D)`, `oct(D)` y `hex(D)` devuelven cadenas de caracteres con la expresión de D en las bases 2, 8 y 16.

Para convertir un entero en base b a decimal podemos usar

$$\text{ZZ}(\text{'expresion en base b'}, b),$$

con el número en base b entre comillas porque debe ser una cadena de caracteres. Entonces, por ejemplo, `ZZ(hex(D),16)` debe devolver D .

5.1.3. La prueba del 9 y otros trucos similares

En el sistema de numeración decimal es fácil calcular el resto de la división de un entero entre 9 sin hacer la división. Ocurre que todas las potencias de 10 tienen resto uno al dividir entre 9 (comprobarlo). Entonces, el resto de dividir entre 9 el entero D es el mismo que el resto de dividir entre 9 la suma de las cifras decimales de D . Podemos así calcular ese resto sumando las cifras y cada vez que superamos 9, volver a sumar las cifras de esa suma, hasta que dejemos de superar 9:

```
k=2312348912498
suma=sum(k.digits())
while suma>9:
    print suma,
    suma=sum(suma.digits())
suma, suma==k%9
```

```
56 11
(2, True)
```

Cualquier operación aritmética entre enteros, por ejemplo una división con resto $D = d \cdot c + r$, se puede comprobar rápidamente cambiando cada entero z por su resto $[z]$ al dividir entre 9 (“tomando clases de restos módulo 9”) para obtener que debe ser $[D] = [d] \cdot [c] + [r]$. Si esta última igualdad no se cumple podemos asegurar que la división está mal hecha, mientras que el recíproco no es siempre cierto.

El mismo argumento nos permite calcular el resto de la división de un entero D entre 3 (o 9) como el resto de la división entre 3 (o 9) de la suma de sus cifras.

5.2. Trucos

En esta sección se recogen algunos *trucos* de programación útiles, es decir reutilizables, y, también, hay enlaces a otras zonas del documento en las que se pueden encontrar otros.

En la hoja de Sage [51-COMPL-trucos.ipynb](#) puedes ver algunos ejemplos relacionados con esta sección.

5.2.1. Potencias

Queremos calcular, en la forma más eficiente posible, una cierta potencia a^n de un entero a . Multiplicando sin más efectuaríamos $n - 1$ productos, y querríamos reducir el número de productos para tratar de acortar el tiempo de cálculo.

Una forma de hacerlo es expresando el exponente en “base 2”, es decir, como una suma de potencias de 2, para obtener

$$a^n = a^{2^{n_0} + 2^{n_1} + \dots + 2^{n_k}} = a^{2^{n_0}} \cdot a^{2^{n_1}} \dots a^{2^{n_k}}.$$

Una vez que hayamos calculado las potencias $a^{2^{n_i}}$ bastará efectuar $k - 1$ productos para terminar el cálculo.

¿Cómo calcular de manera eficiente potencias de la forma a^{2^k} ?

Basta observar que

$$a^{2^k} = (((((a^2)^2)^2) \dots)^2)^2,$$

donde el número de veces que se eleva al cuadrado es exactamente k . En consecuencia, podemos calcular a^{2^k} con sólo k productos: el primero para elevar a al cuadrado, el segundo para elevar a^2 al cuadrado, etc.

El número total de multiplicaciones para calcular a^n sería $(\sum_{i=0}^k n_i) + k - 1$. El número máximo de multiplicaciones que tenemos que hacer, suponiendo que el exponente n es un número de N bits (i.e. se expresa en base dos con N ceros o unos), es el que corresponde al del n que se expresa en base 2 como N unos ($n = 2^N - 1$), y resulta ser

$$0 + 1 + 2 + 3 + \dots + (N - 1) + N - 1 = \frac{N(N - 1)}{2} + N - 1.$$

Esta idea puede expresarse mediante un programa recursivo bastante sencillo

```
def potencia(a,k):
    if k==0:
        return 1
    elif k %2 == 0:
        b = potencia(a,k/2)
        return (b*b)
    else:
```

```

b = potencia(a,(k-1)/2)
return (a*b*b)

```

Si el exponente k es una potencia de 2 el programa nunca entra en el **else**, y vemos que este programa recursivo está implementando las ideas de la discusión anterior.

Es fácil comprobar que este programa es muchísimo más eficiente que uno que fuera acumulando los productos parciales en una variable, y en cada vuelta del bucle multiplicara el valor acumulado por a . Por ejemplo, este programa puede, en un ordenador de sobremesa estándar, elevar 77 al exponente 2^{30} en 152 segundos, mientras que el que usa el acumulador no terminó el cálculo en 12 horas. El motivo tiene que ser que el algoritmo para multiplicar números grandes (¿cuál es?) es muy eficiente de forma que es muchísimo mejor hacer pocas multiplicaciones, aunque sean de números muy grandes, que hacer muchas multiplicaciones, del orden de mil millones, de un número grande, el que está en la variable que va acumulando los resultados parciales, por uno pequeño como 77.

El algoritmo es especialmente eficiente cuando lo que nos interesa es calcular, sin calcular a^k , el resto ($a^k \% m$) de la división de a^k entre un entero m . En ese caso podemos efectuar todas las operaciones *módulo* m , es decir, cada vez que al operar superamos m podemos quedarnos con el resto de dividir entre m .

```

def potencia_mod(a,k,m):
    if k==0:
        return 1
    elif k%2 == 0:
        b = potencia_mod(a,k/2,m)
        return (b*b)%m
    else:
        b = potencia_mod(a,(k-1)/2,m)
        return (a*b*b)%m

```

Esta *aritmética módulo* m no sobrecarga tanto los recursos del ordenador como la aritmética entera con números grandes, y así es posible calcular en 176 segundos

potencia_mod(7777⁽¹²³⁴⁾,2⁽¹⁵⁷⁾,10991⁽⁹⁸⁷⁶⁵⁴⁾+1)

en el que elevamos un entero que tiene unas 4800 cifras decimales a otro que tiene 47 módulo un entero que tiene unos cuatro millones de cifras. ¿Cómo se calcula el número de cifras decimales que tiene un número de la forma a^n ? ¿Cuántas cifras

decimales tiene el entero

$$(7777^{1234})^{2^{157}}?$$

En Sage podemos usar la instrucción **power_mod**(a,n,m) para calcular, por un procedimiento que debe ser¹ el mismo que usa **potencia_mod**(a,n,m), el resto de dividir a^n entre m .

Esta operación, elevar un entero grande a un exponente grande módulo otro entero también grande, es, como veremos en el capítulo 8, usado en criptografía para encriptar mensajes, mientras que la operación inversa, el *logaritmo discreto*, es decir el cálculo del exponente n tal que a^n módulo m es un entero prefijado M , es clave para romper el sistema que encripta mediante potencias.

5.2.2. Listas binarias

Supongamos que queremos iterar sobre todas las listas binarias, ceros o unos, de longitud k . Sabemos que el número de tales listas es 2^k . Una manera es *anidar* k bucles **for** con la variable en cada uno de ellos tomando valor 0 ó 1. Ésto no es práctico si k es grande, y no es factible si queremos que k sea un parámetro en una función de Sage.

Una alternativa es generar la lista de todas las listas binarias de longitud k e iterar sobre ella, lo que se puede hacer fácilmente usando el método **digits**(base=2).

Concretamente, se puede usar un código como el siguiente

```
def funcion(k):
    L = []
    for m in xrange(2^k):
        L.append(m.digits(base=2,padto=k))
    for L1 in L:
        ....
        ....
```

donde el primer bucle genera la lista de todas las listas binarias de longitud k y el segundo itera sobre los elementos de esa lista un bloque de instrucciones que no se ha escrito.

¿Para qué nos puede servir este truco? Intenta escribir, sistemáticamente, todos los polinomios de grado $< k$ con coeficientes en $\mathbb{Z}_j = \{0, 1, 2, \dots, j-1\}$ sin usar el truco y usándolo.

¹Tarda prácticamente lo mismo en hacer el cálculo anterior.

5.2.3. Archivos binarios

Un archivo de texto con N caracteres ocupa $8N$ bits, ceros o unos, o, lo que es lo mismo, N bytes. Cada caracter se codifica como una cadena de 8 bits usando el [código ASCII](#).

Entonces, si generamos una cadena binaria de N bits, N ceros o unos, y la guardamos en un archivo obtendremos uno de $8N$ bits (N bytes), ya que los ceros o unos los trata como un caracter cualquiera, el cero se representa en binario como 00110000 y el uno como 00110001.

Para comprimir archivos, con métodos como `.zip`, necesitamos que un archivo que contenga N bits ocupe N bits, y no $8N$ bits. El truco para conseguir esto consiste en dividir el archivo en trozos de 8 bits y escribir al archivo NO LOS 8 CEROS O UNOS sino el caracter que corresponde a los 8 bits en ASCII. De esta forma lo que realmente se escribe al archivo es la cadena de bits original y el archivo resultante ocupa N bits o $N/8$ bytes.

Ejercicio 1.

1. Define funciones que permitan crear un archivo que contenga la información correspondiente a 2^n bits aleatorios (probabilidad $1/2$ de cero y probabilidad $1/2$ de uno) y que ocupe $2^n/8$ bytes (2^n bits).
2. Dado un real $0 < p < 1$ define una función para generar archivos parecidos a los del punto anterior pero con probabilidad p de cero y $1-p$ de uno. Para hacer esto generamos un decimal aleatorio del intervalo $(0, 1)$ (se puede hacer con `random()`) y si es menor que p producimos un cero y si es mayor un uno.
3. Genera 21 archivos correspondientes a valores de p de la forma $j * (1/20)$ con $0 \leq j \leq 20$, asignándoles automáticamente nombres que incluyan el valor de p . Todos estos archivos deben tener tamaño $2048 * 2048 = 2^{22}$ bits.
4. Comprime los 21 archivos generados

```
tar cvzf <nombre-del-archivo.tar.gz> <nombre-del-archivo>
```

y produce la lista de duplas (p, t_p) con t_p el tamaños en bits del archivo comprimido correspondiente a la probabilidad p .

5. Representa en un gráfico los puntos (p, t_p) , y comprueba así que LOS ARCHIVOS SE COMPRIMEN MÁS CUANTO MÁS PREDECIBLE ES SU CONTENIDO.

5.2.4. La Enciclopedia de sucesiones de enteros

La *Enciclopedia de sucesiones de enteros* es un *lugar web*² que permite acceder y buscar dentro de una inmensa base de datos de sucesiones de números enteros.

En particular, permite, en ocasiones, identificar números reales de los que hemos calculado las primeras cifras decimales usando el ordenador, sin más que escribir las cifras decimales que creemos seguras del número real buscado, separadas por comas, en la ventana que encontramos en la página de acceso.

La Enciclopedia permite identificar sucesiones de enteros, no sólo de dígitos, como por ejemplo, la sucesión de Fibonacci y es enormemente útil al realizar *experimentos matemáticos*.

5.2.5. Enlaces a otras zonas del documento

1. Usando **conjuntos** podemos eliminar elementos repetidos en una lista: primero convertimos la lista L en conjunto mediante $A=\text{set}(L)$, y a continuación el conjunto en lista mediante $L1=\text{list}(A)$.
2. Veremos ejemplos en los que es rentable calcular un número entero muy grande pasando a través de los números reales. Uno típico se encuentra al calcular un número de Fibonacci muy grande sin calcular los anteriores (ver la sección 6.3).
3. Usando logaritmos podemos calcular, ver la sección 7.5, el número de cifras que tendrá en una base de numeración dada b un número de la forma a^k , sin necesidad de calcular a^k . También usando logaritmos es posible calcular la *cifra dominante* de un entero de la forma a^k , es decir, su primer dígito por la izquierda, sin necesidad de calcular a^k .
4. Por supuesto, podemos calcular de manera eficiente la cifra de las unidades de a^k , en una base cualquiera b mediante **potencia.mod**(a,k,b) como hemos visto en esta sección. ¿Cómo podríamos calcular la segunda cifra por la derecha o la tercera?
5. Hemos visto al menos dos ejemplos, **mergesort**, y en la definición de la función potencia en esta misma sección, en los que el problema se presta a un tratamiento recursivo especialmente eficiente.

Esto ocurre porque en cada llamada recursiva a la propia función se divide el tamaño de los datos por dos, y se consigue así que la profundidad del árbol recursivo de llamadas sea del orden de $\log(n, \text{base}=2)$ con n el entero que “mide el tamaño

²Originalmente se publicó en papel.

de los datos”, por ejemplo, si el dato es una lista, como en mergesort, n sería su longitud, y en la función potencia n sería el exponente que, en este caso, también hemos llamado n .

6. En la sección del capítulo 3 dedicada a los diccionarios se explica cómo se pueden usar los diccionarios de Sage para estructurar la información de forma que las búsquedas sean rápidas. El procedimiento es similar a la ordenación *lexicográfica* de las palabras en un diccionario de papel, que es lo que nos permite encontrar una palabra concreta rápidamente.
7. ¿CONTINUARÁ?

5.3. Fuerza bruta

En muchos problemas matemáticos tenemos que encontrar, en un cierto conjunto X , un elemento x que cumple una cierta propiedad P .

Si X es finito, podemos construirlo en el ordenador, y resolver el problema recorriendo el conjunto hasta que encontremos un elemento que cumple P . Decimos que este método para resolver el problema es de *fuerza bruta*.

Por ejemplo, en criptografía podemos haber interceptado un mensaje que sabemos que ha sido encriptado con un cierto método que conocemos. Entonces “conocemos” el conjunto finito de todas las posibles claves para desencriptarlo y, en teoría, podemos ir probando todas hasta que encontremos una clave tal que al desencriptar con ella se obtenga un mensaje legible. Es claro que cualquier sistema criptográfico debe tener un conjunto de claves posibles tan grande que sea imposible desencriptar mediante fuerza bruta. Este método puede funcionar en algunos casos, pero, como veremos hay sistemas criptográficos que son inmunes a él.

Como método es claro que es bastante “bruto”, no tenemos que pensar mucho para aplicarlo, y, por otra parte, está limitado por el tamaño máximo de los X que podamos manejar en nuestro ordenador. Puede ser que el X que nos interese sea demasiado grande para que pueda caber en la memoria RAM, o bien que el proceso de comprobar si x cumple la propiedad P requiera demasiado tiempo, de forma que el tiempo total de cálculo pudiera ser demasiado grande, meses o años, para que nos interese estudiar el problema así.

El problema de la RAM se resuelve en parte usando **iteradores**, que recorren los elementos del conjunto X “uno a uno” sin construir en RAM el conjunto completo. Si es posible construir los elementos de X usando iteradores nos quedaría, en muchos casos, sólo el segundo problema cuya solución es armarnos de paciencia o bien tratar de *paralelizar* el problema.

En los ordenadores que estamos usando, mi experiencia es que suele ser viable tratar estructuras de datos, por ejemplo listas, de un tamaño del orden de 10^6 , pero suelen aparecer problemas cuando se llega a 10^7 .

5.4. Cálculo en paralelo

Las máquinas que tenemos actualmente en el Laboratorio disponen de un procesador con dos núcleos (2 *cores*), lo que significa que *en teoría* pueden realizar cálculos usando los dos al mismo tiempo en la mitad del tiempo. De hecho, gracias a una técnica llamada *hyperthreading*, el sistema operativo de la máquina cree que tiene cuatro núcleos con cada núcleo físico soportando dos virtuales, pero, como debemos esperar, los tiempos que se obtienen calculando con cuatro son prácticamente iguales a los que se obtienen con dos.

Veremos al menos un par de ejemplos de cálculo en paralelo, uno en un ejemplo de ataque mediante fuerza bruta de un sistema criptográfico (capítulo 8) y el otro el cálculo aproximado de áreas planas usando puntos aleatorios (capítulo 11).

Es claro que la mayor parte de los programas pasan casi todo su tiempo ejecutando bucles y podemos paralelizar un bucle dividiendo su rango en tantos trozos como núcleos y mandando un trozo a cada núcleo. Sin embargo:

1. Al dividir el trabajo entre los núcleos es posible, dependiendo del problema, que varios necesiten los mismos datos, o bien que un núcleo necesite en algún momento resultados que calcula otro. Aparece entonces un problema *logístico* serio: los núcleos pueden tener que comunicarse entre ellos, y si se organiza mal es posible que haya muchos núcleos esperando que les llegue información que necesitan para continuar su trozo del cálculo.
2. En máquinas *multicore* el problema de la comunicación se resuelve usando zonas de memoria RAM compartida por todos los núcleos (*shared memory*). El *software* estándar para controlar el acceso a la memoria compartida se llama **OpenMP**.
3. También se puede hacer cálculo paralelo en *clusters* de ordenadores que se comunican enviándose mensajes a través de la red local. Esto es bastante más complicado que el caso anterior, y se hace usando un protocolo llamado **MPI** del que existen varias implementaciones.
4. Hay bucles completamente *imparalelizables*, por ejemplo, si cada vuelta del bucle necesita el resultado de la anterior. ¿Se te ocurre algún ejemplo?

Si dividimos el trabajo entre los núcleos, en uno de esos casos imparalelizables, puede ocurrir que sólo uno pueda calcular en cada momento y los otros van a estar esperando a que les llegue su turno. El tiempo de ejecución va a ser igual o mayor que si ejecutamos el bucle en un sólo núcleo.

5. Algoritmos matemáticos complejos pueden ser muy difíciles de paralelizar, de manera que sistemas como Sage utilizan casi siempre un único núcleo. Si el cálculo es muy largo podemos observar, usando el *System monitor*³, que el núcleo en uso va cambiando de tiempo en tiempo. Eso lo hace el sistema operativo, no Sage, para evitar el sobrecalentamiento del procesador.
6. Bucles que pueden ser divididos en trozos completamente independientes pueden siempre ejecutarse en paralelo, pero es conocido como *paralelismo vergonzante* para indicar que para practicarlo no es necesario pensar mucho.
Sage dispone de una implementación de esta clase de paralelismo y es lo único que veremos en este curso sobre el asunto.
7. Incluso el paralelismo vergonzante tiene un problema: el tiempo que tarda uno de los núcleos en ejecutar una vuelta del bucle puede depender del tamaño de los datos, y en tales casos el número de vueltas que asignamos a cada núcleo puede ser distinto ya que tenemos que *equilibrar la carga que soporta cada núcleo*. Veremos algún ejemplo de esta situación a continuación, junto con un truco que permite, en ocasiones, equilibrar la carga fácilmente.

Consideremos, por ejemplo un problema en el que debemos elevar al cuadrado un gran número de matrices, de tamaño creciente m , con entradas enteros aleatoriamente elegidos. Haciendo el cálculo en un único núcleo obtenemos, por ejemplo, un tiempo

```
def elevar2_m(m):
    L = [randint(0,1000) for i in xrange(m*m)]
    M = matrix(ZZ,m,m,L)
    return M*M
```

```
time LL = map(elevar2_m,xrange(1,200))
```

Time: CPU 22.98 s, Wall: 23.27 s

En este ejemplo, el tamaño de las matrices varía entre 1×1 y 199×199 . En una máquina con 4 núcleos podemos utilizarlos todos mediante

```
@parallel(4)
def cuadrado(L):
```

³Aplicaciones/Herramientas del sistema/Monitor del sistema


```

    map(elevar2_m,L)
time LL =
list(cuadrado([srange(1,50),srange(50,100),srange(100,150),srange(150,200)]))

```

Time: CPU 0.13 s, Wall: 14.79 s

En este reparto de la carga el primer núcleo tiene que tratar con matrices de tamaño mucho menor y, por tanto, acabará antes su tarea. La carga está muy desequilibrada. El tiempo de cálculo es el que se obtiene para el núcleo que más tarda, es decir el que calcula con tamaños entre 150 y 200.

```

@parallel(4)
def cuadrado(L):
    map(elevar2_m,L)
L1 = [m for m in xrange(1,200) if m%4 == 0]
L2 = [m for m in xrange(1,200) if m%4 == 1]
L3 = [m for m in xrange(1,200) if m%4 == 2]
L4 = [m for m in xrange(1,200) if m%4 == 3]
time LL = list(cuadrado([L1,L2,L3,L4]))

```

Time: CPU 0.02 s, Wall: 5.82 s

Mediante este truco conseguimos que la carga se equilibre, los cuatro núcleos tardan más o menos lo mismo y ese es el tiempo total resultante.

Puedes comprobar lo anterior en la hoja [52-COMPL-paralelo.ipynb](#). Los tiempos por supuesto dependen del ordenador que estemos usando.

5.5. Eficiencia

Debemos tratar de escribir programas SINTÁCTICAMENTE CORRECTOS, es decir, que el intérprete de Python acepte como válidos y no muestre mensajes de error, y SEMÁNTICAMENTE CORRECTOS, es decir, que calculen lo que pretendemos calcular. Es claro que un programa que no cumple estas dos condiciones no nos sirve.

Pero además, nuestros programas deben ser EFICIENTES, no deben tardar más tiempo del necesario ni utilizar más memoria RAM de la necesaria. En esta sección discutimos algunos aspectos de la posible mejora en la eficiencia de nuestros programas.

Una de las reglas básicas, que ya discutimos en la sección 4.2.2, es que DEBEMOS CALCULAR EL MÍNIMO NECESARIO PARA PODER RESPONDER a la pregunta que nos hacemos: si queremos calcular la longitud de una lista en lugar de calcular la lista completa y, una vez está en memoria, calcular su longitud debemos usar un “contador” que se incrementa según vamos calculando elementos que deberíamos añadir a la lista pero sin generar la lista.

De la misma forma, si queremos calcular la suma (producto) de una serie de números es mejor irlos sumando (multiplicando) en un “acumulador” que generar la lista completa y luego sumar (multiplicar).

5.5.1. Control del tiempo

Para empezar, podemos usar la instrucción **time** al comienzo de una línea de código en la que se ejecute una función propia de Sage o definida por nosotros. Al mostrarnos la respuesta también muestra dos tiempos: el tiempo de CPU y el llamado *Wall time* que corresponde al tiempo transcurrido desde que empezó el cálculo hasta que terminó. No coinciden necesariamente, aunque si somos el único usuario y no estamos ejecutando otros programas aparte de Sage prácticamente coincidirán, porque el ordenador puede dedicar algo del tiempo de CPU a otras labores no relacionadas con nuestro cálculo.

En segundo lugar tenemos la instrucción **timeit**, que se ejecuta en la forma **timeit('instruccion')**, y difiere de **time** en que ejecuta la instrucción varias veces y devuelve un promedio. La instrucción admite varios parámetros (averiguar con la ayuda interactiva para **timeit**) que controlan el número de repeticiones, etc.

Por último, **prun** produce un resultado mucho más detallado que nos permite saber cómo se distribuye el tiempo total de cálculo entre las distintas funciones que se ejecutan dentro de nuestro programa.

Se invoca el **profiler** mediante

```
% prun -q -T <archivo.txt> <instruccion>
```

y hay un par de ejemplos de su uso en la hoja de Sage [53-COMPL-eficiencia-tiempo.ipynb](#) que acompaña esta subsección.

Hay otro **profiler**, **lprun**, para código **Python**, no funciona sobre funciones que contienen código Sage, que analiza el tiempo total utilizado en cada línea de código.

```
%load_ext line_profiler
%lprun -q -T <archivo.txt> -f <funcion> <funcion>(<parametros>)
```

5.5.2. Control de la RAM

Programas que construyen grandes estructuras de datos, por ejemplo listas enormes, pueden llegar a saturar la memoria RAM de la máquina en la que estemos trabajando. Se pueden ver los incrementos en el uso de RAM mediante la instrucción `get_memory_usage()` que nos devuelve la cantidad de memoria RAM, en megabytes (MB), que está en uso en el momento en que se invoca.

Usándola, como en la hoja [54-COMPL-eficiencia-ram.ipynb](#) que acompaña esta subsección, podemos ver los incrementos en memoria RAM al generar grandes estructuras de datos.

Para funciones que usan **Python**, pero no Sage, disponemos de un **profiler** de memoria RAM, **mprun**, que funciona de forma parecida a **lprun**.

```
%load_ext memory_profiler
%mprun -T <archivo> -f <funcion> <funcion>(<parametros>)
```

Si hemos ejecutado código previamente en la hoja hay que reiniciar el núcleo para que libere toda la memoria utilizada previamente. Si no se hace se acumula a la ya utilizada.

Pueden verse ejemplos de uso de **lprun** y **mprun** en la hoja [55-COMPL-lprun-mprun.ipynb](#).

5.5.2.1. Iteradores

1. Muchas estructuras de datos son *iterables*, es decir, podemos crear un bucle **for** que recorra uno por uno los elementos de la estructura de datos y para cada uno de esos elementos ejecute un bloque de instrucciones. Todas las estructuras de datos básicas, que vimos en el capítulo 3, listas, tuplas, cadenas de caracteres, conjuntos y diccionarios, son iterables.

Sin embargo, esta forma de iterar funciona creando en memoria la estructura de datos completa y, a continuación, recorriéndola. Si, por ejemplo, la lista es enorme va a ocupar una gran cantidad de memoria RAM, y crearla y recorrerla puede ser un procedimiento muy ineficiente.

2. Estos inconvenientes se resuelven en parte con los *iteradores* o *generadores*, que en lugar de crear la estructura de datos en memoria para luego iterar sobre ella, van generando elementos de uno en uno y cada vez que tienen uno ejecutan el bloque de instrucciones del bucle sobre él.

Es claro que esta forma de iterar debe ser mucho más eficiente, al menos en términos de memoria RAM.

3. El iterador básico es `xsrangle(k)` que genera enteros de la lista `srangle(k)` de uno en uno. Si por ejemplo definimos `gen = xsrangle(10^8)`, no se crea la lista sino únicamente el generador `gen`, y ejecutando varias veces `gen.next()` podemos ir viendo los enteros sucesivos `0, 1, 2, ...`.

Iteramos sobre un generador de la misma manera que sobre una lista: `for j in xsrangle(10^7):...`

4. Es posible crear nuevos generadores usando la sintaxis breve para bucles:

```
gen2 = (f(x) for x in xsrangle(10^8) if Condition)
```

que produce un generador nuevo, a partir del básico `xsrangle(10^8)`, y filtrado por la condición booleana `Condition`. Nótese que el generador va delimitado por paréntesis en lugar de corchetes.

5. Por ejemplo

```
g = (n**2 for n in xsrangle(10**8) if is_prime(x))
```

define un generador que produce los cuadrados de los primos, *bajo demanda*, es decir, cada vez que ejecutamos `g.next()` devuelve el cuadrado de un primo.

5.5.3. Cython

Python no se compila al lenguaje de máquina, sino que se interpreta (i.e. se va traduciendo a lenguaje de máquina sobre la marcha al irse ejecutando), lo que hace que, en general, código escrito en Python se ejecute mucho más lentamente que código escrito en lenguajes que se compilan.

Sin embargo, modificando muy poco código escrito en Python es posible compilarlo automáticamente a través de C, con lo que se consiguen mejoras impresionantes en su eficiencia. Cuando un programa se compila, por ejemplo en C, es necesario que el código reserve explícitamente la memoria que se va a utilizar durante el cálculo. Para eso existen *tipos de datos* predefinidos que ocupan cantidades prefijadas de memoria RAM.

cython traduce código escrito en Python a C y, una vez que se declaran los tipos de las variables, consigue mejoras importantes en el rendimiento. Para usarlo dentro de Sage basta escribir en la primera línea de la celda `%cython` y declarar los tipos de las variables. Por ejemplo

```
%cython
def cuadrado(double x):
    cdef int a=1
    return x*x+a
```

define una función de **cython** en la que x está declarada como un real de doble precisión y a como un entero. Los tipos más usados son **int** para enteros, **long** para enteros grandes, **float** para decimales y **double** para decimales de precisión doble.

Cuando el programa incluye además listas o matrices grandes es conveniente, y se consiguen enormes mejoras adicionales, usar los correspondientes objetos de **numpy** en lugar de los propios de Sage.

Pueden verse algunos ejemplos de estas mejoras en la hoja [56-COMPL-cython.ipynb](#), donde se discute la manera de conseguir grandes cantidades de números aleatorios de manera eficiente, es decir, usando librerías de funciones compiladas en C. Volveremos sobre este asunto en el capítulo 11, Probabilidad, en el que será crucial.

En la hoja [57-COMPL-planeta-cython-sympl.ipynb](#) se simula, usando métodos elementales, el movimiento de un planeta alrededor del Sol de acuerdo a las Leyes de Newton. Si te interesan los detalles de este asunto puedes leer, y es absolutamente recomendable, este [capítulo en el libro de Física de Feynman](#).

5.5.4. Numpy

Python dispone de un número grande de módulos adicionales que amplían sus capacidades y resuelven algunos de sus problemas. **numpy** está diseñado para permitir operaciones muy rápidas con listas y matrices, de hecho con *arrays* de cualquier dimensión, y permite, en conjunción con Cython, utilizar Python (y Sage), en Cálculo Numérico.

En este curso veremos muy poco Cálculo Numérico ya que hay una asignatura específica en el segundo cuatrimestre, y además en ella se utiliza Matlab en lugar de Sage, pero sería perfectamente factible utilizar la combinación Sage+**cython**+**numpy** para conseguir los mismos resultados que con Matlab⁴ en cuanto a eficiencia.

Para poder usar **numpy** dentro de Sage debemos evaluar una celda con el contenido **import numpy as np**, y una vez hecho esto una línea como $A = \text{np.array}([1,2],[1,1])$ define la matriz A como una matriz de **numpy**. Una vez que definimos una lista o matriz como de **numpy**, las operaciones que hagamos con ella serán operaciones de **numpy** y, por tanto, muy optimizadas.

⁴Matlab, al contrario que Sage, no es un programa gratuito sino bastante caro y es la Universidad quien paga las licencias que nos permiten usarlo en el Laboratorio. Además, el motor de cálculo de Matlab consiste en librerías para Álgebra Lineal, muy optimizadas, que están en el dominio público hace mucho tiempo.

La sintaxis de **numpy** es algo diferente a la de Sage, por ejemplo $A * A$ para matrices de **numpy** es el producto elemento a elemento mientras que el producto de matrices se obtiene mediante $A.\text{dot}(A)$. En este curso no usaremos sistemáticamente, aunque quizá sí puntualmente, **numpy** para el cálculo con matrices.

En la hoja [58-COMPL-numpy.ipynb](#) se pueden ver un par de ejemplos que comparan los tiempos de ejecución de códigos que utilizan **numpy** con códigos similares que no lo utilizan.

Un ejemplo más se encuentra en la hoja

[59-COMPL-fractal-cython-numpy-MV.ipynb](#)

dedicada a la obtención de gráficas de conjuntos fractales, y en particular del conjunto de Mandelbrot.

Puedes leer la [página de la Wikipedia](#) sobre el conjunto de Mandelbrot y [esta otra](#) sobre la noción general de conjunto fractal. Además en la wiki de Sagemath puede encontrarse [esta página](#), en la que hay varios ejemplos de construcción de fractales con Sage, junto con ejemplos sobre cómo hacer las gráficas interactivas.

Parte II

Aplicaciones

Capítulo 6

Teoría de números

La *teoría de números* o *aritmética* estudia las propiedades los números enteros y de algunos otros conjuntos de números que generalizan a los enteros.

Una de las características de la teoría de números es la existencia de una gran cantidad de PROBLEMAS DE ENUNCIADO MUY SENCILLO¹ PERO DE DIFÍCIL, O DESCONOCIDA, DEMOSTRACIÓN. En todos estos casos se han hecho comprobaciones masivas mediante ordenador sin encontrar un contraejemplo, de forma que se cree que los resultados son ciertos pero en muchos de ellos se cree también que la demostración está todavía lejos.

Muchos de los ejercicios planteados en este capítulo son *ejercicios retóricos*, es decir, son ejercicios de programación, que como tales son útiles para aprender a programar, pero que no nos dicen gran cosa sobre los problemas matemáticos, extremadamente difíciles, que estudian.

En otros casos plantearemos algunos ejercicios cuya respuesta ya debe ser conocida, teóricamente, a través de la asignatura *Conjuntos y Números*, y por tanto son también, en otro sentido, *retóricos*. En particular, supondremos conocidos, aunque los repasaremos brevemente, los siguientes contenidos:

1. Las nociones de grupo abeliano, anillo y cuerpo. Las definiciones básicas acerca del anillo de los números enteros y el cuerpo de los racionales. El teorema de factorización de enteros como producto de primos, a veces llamado *teorema*

¹Para muchos de ellos basta con conocer *las cuatro reglas*.

fundamental de la aritmética.

2. La definición de los *anillos \mathbb{Z}_m de clases de restos módulo un entero m* y el hecho de que son cuerpos únicamente aquellos en los que el módulo es primo.
3. Las nociones de *máximo común divisor (MCD)* y *mínimo común múltiplo (MCM)*, el algoritmo de Euclides para calcular el MCD y su consecuencia, *el teorema de Bézout*. Este último resultado lo usamos para calcular el inverso de un elemento invertible en un anillo de clases de restos y, por tanto, es importante.
4. El *teorema pequeño de Fermat*, y su generalización, *el teorema de Fermat-Euler*, que estudian las potencias de elementos en un anillo de clases de restos. En el enunciado del teorema de Fermat-Euler aparece la función ϕ de Euler, que también supondremos conocida.

Estos resultados serán utilizados más adelante en criptografía y al estudiar algunos criterios para determinar si un entero es primo.

6.1. Grupos, anillos y cuerpos

Llamamos Álgebra al estudio de las estructuras algebraicas, es decir, conjuntos con operaciones que cumplen ciertas propiedades. Este estudio incluye el de los sistemas de ecuaciones polinomiales en una o varias variables, que era el objeto del Álgebra clásica.

6.1.1. Grupos

Un **grupo** es un conjunto G con una operación, que denotamos mediante $g * g'$, definida para cada dos elementos del grupo y que como resultado da un elemento del grupo, verificando

1. (Asociatividad) $(g * g') * g'' = g * (g') * g''$ para cualesquiera elementos del grupo.
2. (Elemento neutro) Existe un elemento $e \in G$ tal que $g * e = e * g = g$ para cualquier $g \in G$.
3. (Inverso) Para todo elemento $g \in G$ existe otro, denotado mediante g^{-1} , tal que $g * g^{-1} = g^{-1} * g = e$.

1. Si para todo par de elementos, $g, g' \in G$, se verifica $g * g' = g' * g$ decimos que el grupo es **conmutativo** o también **abeliano**.
2. Ejemplos básicos de grupos son los números enteros (rationales, reales o complejos) con la operación *suma* y los racionales (reales o complejos) no nulos con la operación *producto*, pero los que más nos interesan en este curso son los asociados a las clases de restos.
3. El número de elementos de un grupo G que sea finito se llama su **orden** ($\text{ord}(G)$).
4. Un **subgrupo** de un grupo G es un subconjunto G_1 de G tal que con la misma operación de G es un grupo. En particular es necesario y suficiente para que G_1 sea un subgrupo que, para cada dos elementos g, g' , de G_1 , $g * (g')^{-1}$ también esté en G_1 .
5. Denotamos mediante g^n el resultado de $g * g * g \dots n$ **factores** $\dots * g$. No hace falta escribir paréntesis gracias a la propiedad asociativa. De la misma forma g^{-n} es $(g^{-1})^n$.
6. Todo elemento $g \in G$, genera un subgrupo $\langle g \rangle$ de G definido como

$$\langle g \rangle := \{g^n \mid n \in \mathbb{Z}\}.$$

7. Si G es finito necesariamente $\langle g \rangle$ también lo será, y definimos el **orden** del elemento g , $\text{ord}(g)$, como el orden del subgrupo $\langle g \rangle$ generado por g .
8. La propiedad fundamental de estos *órdenes* es el teorema de Lagrange afirmando que *el orden de todo subgrupo, en particular el orden de todo elemento, divide al orden del grupo*.

Dado el subgrupo $G_1 \subset G$, el teorema de Lagrange se cumple gracias a que las clases de equivalencia determinadas en G por la relación $g \sim g'$ si $g * (g')^{-1}$ pertenece a G_1 tienen todas el mismo número de elementos que G_1 .

6.1.2. Anillos

Un **anillo** es un conjunto A con dos operaciones, que denotamos habitualmente como suma y producto, tal que

1. Con respecto a la suma A es un grupo abeliano.

2. (Neutro para el producto) Existe un elemento que denotamos mediante 1 en A tal que $a \cdot 1 = 1 \cdot a = a$ para todo elemento $a \in A$.
3. (Distributiva) Para toda tripleta de elementos $a, a_1, a_2 \in A$ se verifica $a \cdot (a_1 + a_2) = a \cdot a_1 + a \cdot a_2$.

Si el producto de dos elementos cualesquiera es conmutativo ($a_1 \cdot a_2 = a_2 \cdot a_1$) decimos que el **anillo es conmutativo**.

Los números enteros, racionales, reales y complejos son anillos conmutativos con las operaciones habituales. Las matrices cuadradas $n \times n$ son un anillo únicamente conmutativo en el caso $n = 1$. Los anillos que más nos interesan son los de clases de restos.

6.1.3. Cuerpos

Un **cuerpo** K es un anillo en el que todo elemento no nulo tiene un inverso para el producto, es decir, para todo $x \neq 0$ existe un elemento x^{-1} tal que $x \cdot x^{-1} = x^{-1} \cdot x = 1$.

Los anillos de racionales, reales y complejos son cuerpos, pero los que más nos interesan son los de clases de restos módulo un primo.

6.2. Clase de restos

Fijamos un entero m al que llamamos *módulo*. En el anillo de los números enteros la relación $nR_m n'$ si y sólo si n y n' tienen el mismo resto al dividir por m es una relación de equivalencia. Denotamos por $[n]_m$ la clase de n módulo m , aunque habitualmente escribimos $[n]$ sin que el módulo aparezca explícitamente en la notación.

El conjunto cociente tiene m elementos que corresponden a los m posibles restos, es decir, una clase, $[0]$, contiene todos aquellos enteros que al dividir entre m dan resto 0 (múltiplos de m), otra, $[1]$, aquellos que dan resto 1, y así hasta $m - 1$.

La observación fundamental es que el conjunto de clases (o conjunto cociente) tiene una estructura natural de anillo:

1. Suma: $[n_1] + [n_2] = [n_1 + n_2]$
2. Producto: $[n_1] \cdot [n_2] = [n_1 \cdot n_2]$

Esta definición de las operaciones es natural: para operar dos clases se eligen elementos **cualquiera** en cada una de ellas, se operan y la clase resultado es la clase del resultado. Es esencial comprobar que la clase del resultado no depende de

la elección de representantes de cada una de las clases, porque en caso contrario la operación no estaría bien definida (habría varios resultados para una misma suma o producto).

Se obtiene así un anillo conmutativo, al que denotamos mediante \mathbb{Z}_m , y llamamos *el anillo de clases de restos módulo m* .

¿En qué condiciones es el anillo \mathbb{Z}_n un cuerpo? Gracias al **teorema de Bezout** y dado que el $MCD(k, p) = 1$ si p es primo y $k < p$, sabemos que existen enteros a y b tales que $a \cdot k + b \cdot p = 1$ de forma que tomando restos módulo p se obtiene que el inverso de $[k]$ es $[a]$.

El cálculo de potencias en los anillos de clases de restos \mathbb{Z}_m es, computacionalmente, muy eficiente ya que podemos efectuar todas las operaciones módulo m . Al discutir un método para calcular **potencias** de elementos de un anillo ya vimos que para los anillos de clases de restos el algoritmo utilizado era especialmente eficiente.

Estos cálculos, potencias de un elemento en un anillo de clases de restos, son, importantes en criptografía, y concretamente en el método llamado RSA que es uno de los más utilizados en la práctica. En el fundamento teórico de la criptografía RSA está el llamado *teorema de Fermat-Euler*, un resultado bastante elemental pero enormemente importante en aritmética y sus aplicaciones.

6.2.1. Teorema de Fermat-Euler

Sea $[a] \in \mathbb{Z}_p$ una clase de restos no nula módulo un entero primo, y definimos la función $M_{[a]} : \mathbb{Z}_p \rightarrow \mathbb{Z}_p$ definida por multiplicación por $[a]$: $M_{[a]}([b]) = [a] \cdot [b]$. Iterando la función $M_{[a]}$, dado que el conjunto de clases de restos es finito, llegaremos, partiendo de una clase $[b]$ no nula cualquiera, necesariamente a un elemento ya visitado, es decir, $M_{[a]^t}([b]) = M_{[a]^s}([b])$ ($[a]^t \cdot [b] = [a]^s \cdot [b]$) con $s < t$. Como $[a]$ es invertible, la igualdad entre paréntesis implica $[a]^{t-s} \cdot [b] = [b]$, y como $[b]$ también es invertible obtenemos finalmente $[a]^{t-s} = [1]$.

En resumen, para todo elemento no nulo $[a] \in \mathbb{Z}_p$ existe un entero positivo $n_{[a]}$ mínimo tal que $[a]^{n_{[a]}} = [1]$. Es claro, entonces, que la iteración de la función $M_{[a]}$ termina siendo la función identidad.

Ejercicio 1.

1. En una hoja de Sage define $p = \text{nth_prime}(2013)$, y escribe un programa para calcular el conjunto formado por los enteros $n_{[a]}$ con $[a]$ recorriendo los elementos no nulos de \mathbb{Z}_p .
2. El mínimo común múltiplo de todos los enteros $n_{[a]}$ es un entero N tal que $[a]^N = [1]$ para toda clase $[a] \neq [0]$. Encuentra, en \mathbb{Z}_p como en el apartado anterior, el mínimo N con esa propiedad.
3. Consideramos ahora un módulo compuesto, por ejemplo

$$m = \text{nth_prime}(2013) * \text{nth_prime}(2014),$$

y queremos estudiar el mismo tipo de resultados.

Observar que las clases invertibles no son ahora todas las no nulas, sino las clases de enteros $[k]$ tales que k es primo con m . ¿Por qué?

¿Cómo se modifican los resultados anteriores, válidos para módulo primo, en el caso de módulo compuesto?

4. Escribe un programa que calcule las clases en \mathbb{Z}_m que son su propio inverso para la multiplicación, es decir, clases $[a]$ tales que $[a] \cdot [a] = [1]$. Utiliza el programa para tratar de caracterizar esas clases en términos de la factorización de m como producto de primos.

Utilizando ahora el lenguaje de la primera sección podemos enunciar

1. Cuando p es un entero primo \mathbb{Z}_p es, gracias al **teorema de Bezout**, un cuerpo. El conjunto de sus elementos no nulos \mathbb{Z}_p^* es un grupo abeliano respecto al producto de orden $p - 1$. El **teorema pequeño de Fermat** afirma que todo elemento en \mathbb{Z}_p^* elevado a $p - 1$ resulta ser $[1]$. Es consecuencia de que, por el de Lagrange, el orden de todo elemento es un divisor de $p - 1$.
2. Para módulos compuestos m , el anillo \mathbb{Z}_m no puede ser un cuerpo porque hay elementos no nulos, las clases de los factores de m , cuyo producto es nulo. Sin embargo, hay un grupo de elementos invertibles, que podemos denotar mediante $U(m)$, formado por todos los elementos de \mathbb{Z}_m que tienen un inverso para el producto y que, al menos, contiene a los elementos $[\pm 1]$.
3. Gracias al **teorema de Bezout** podemos ver que son invertibles las clases de todos los enteros que son primos con el módulo m . Se denota mediante $\phi(m)$ el orden del grupo de $U(m)$ de invertibles módulo m .

El **teorema de Fermat-Euler** afirma que *todo elemento de $U(m)$ elevado al exponente $\phi(m)$ da como resultado $[1]$* , y como el teorema pequeño de Fermat es consecuencia del de Lagrange.

4. El orden del grupo $U(m)$ se puede calcular en Sage mediante **euler_phi**(m), y, si conocemos la factorización de m se puede calcular mediante la fórmula

$$\phi(m) = m \prod_{p|m} \left(1 - \frac{1}{p}\right),$$

con el producto recorriendo los divisores primos p de m .

Hemos utilizado estos resultados al resolver el **ejercicio 8** del capítulo anterior.

6.3. Fibonacci

La sucesión de Fibonacci es la de números enteros definida por

$$F_0 = 0, F_1 = 1, F_m = F_{m-1} + F_{m-2},$$

y forma parte de una gran familia de sucesiones que decimos que están *definidas recursivamente*. En Sage disponemos de la función **fibonacci**(m) que, como es esperable, devuelve el número de Fibonacci m -ésimo. En los primeros ejercicios de esta sección implementamos diversos métodos para calcular F_m y la función **fibonacci**(m) de Sage nos puede servir para comprobar los resultados obtenidos.

Ejercicio 2.

1. Programar una función recursiva que dado un entero m devuelva el m -ésimo número de Fibonacci F_m . Este ejercicio se ha resuelto como un ejemplo en la página **89**.
2. Programar una función iterativa que dado un entero m devuelva el m -ésimo número de Fibonacci F_m . Este ejercicio también se ha resuelto como un ejemplo, ahora en la página **80**.
3. Observa que es fácil obtener el vector $(F_{k+1}, F_k)^t$ multiplicando el par $(F_k, F_{k-1})^t$ por una cierta matriz **A** que no depende de k . Entonces, obtendríamos el vector $(F_{k+1}, F_k)^t$ multiplicando $(1, 0)^t$ por una potencia adecuada de la matriz **A**.

Implementa esta manera de calcular la sucesión de Fibonacci usando el método rápido para calcular potencias (ver p. **118**).

4. Hay fórmulas que permiten calcular el m -ésimo número de Fibonacci F_m sin haber calculado los anteriores. Una tal fórmula, debida a Binet (1843), afirma que

$$F_m := \frac{\phi^m - (1 - \phi)^m}{\sqrt{5}},$$

con ϕ el número áureo $\frac{1+\sqrt{5}}{2}$. Observa que, aunque la parte derecha de la fórmula es una expresión complicada que involucra raíces cuadradas de 5, la fórmula está afirmando que el resultado es un número entero para todo valor de m , es decir, que las raíces se van a cancelar al operar.

En Sage podemos operar de manera exacta, es decir, de forma *simbólica* como opuesta a *numérica*, con expresiones como ϕ . Basta tomar `sqrt(5)`, la raíz cuadrada de 5, y usar las operaciones aritméticas habituales. Se puede forzar a Sage a desarrollar, quitar los paréntesis, una expresión como F_m usando la función `expand` (ver pág. 28).

- Comprueba que F_m es el número de Fibonacci m -ésimo para 20 valores de m mayores que 10000. Estudia la eficiencia de esta fórmula como medio para calcular el número de Fibonacci m -ésimo.
 - Define una función `fibonacci.num(m,d)` que devuelva el número F_m calculado, aproximadamente, como número real con d dígitos de precisión.
 - Mediante las funciones `floor(x)`, `ceil(x)` o `round(x)`, usa la aproximación de F_m dada por la fórmula de Binet para calcular el valor exacto entero de F_m .
 - Estudia la eficiencia de este método, comparando con el método del apartado a (siempre para valores de m mayores que 10000), y los errores que pueden aparecer al usar números reales con d dígitos de precisión.
5. Dado un número de Fibonacci bastante grande F_m queremos calcular el lugar que ocupa en la sucesión, es decir, dado un F_m queremos obtener el m .
- Define, usando la construcción iterativa de la sucesión de Fibonacci, una función que reciba como argumento F_m y devuelva m .
 - Cuando m crece, el segundo sumando (de hecho es un *restando*) en el numerador del F_m del ejercicio anterior tiende a cero. Explica el motivo.
Entonces, F_m es aproximadamente igual a $\phi^m/\sqrt{5}$, y esto podría servir para calcular m dado un F_m suficientemente grande. Implementa un tal método, y estudia su grado de validez comparando los valores de m que produce para variados F_m calculados a partir de emes dados.

6.3.1. Algunas propiedades de la sucesión de Fibonacci

6.3.1.1. Fórmulas

Los números de Fibonacci tienen una enorme cantidad de relaciones entre ellos, y muchas de ellas se pueden demostrar por inducción. Algunas de las más simples son las siguientes:

1.
$$F_0 + F_1 + F_2 + \cdots + F_m = F_{m+2} - 1,$$
2.
$$F_1 + F_3 + F_5 + \cdots + F_{2m-1} = F_{2m},$$
3.
$$F_0 - F_1 + F_2 - F_3 \cdots - F_{2m-1} + F_{2m} = F_{2m-1} - 1,$$
4.
$$F_0^2 + F_1^2 + F_2^2 + \cdots + F_m^2 = F_m F_{m+1},$$
5.
$$F_{m-1} F_{m+1} - F_m^2 = (-1)^m.$$

Ejercicio 3. *Es fácil, y debes hacerlo, escribir programas que comprueben, hasta un m muy grande, que cualquiera de estas fórmulas es correcta, pero lo que sería más interesante es encontrar métodos para generar automáticamente y demostrar fórmulas de esta clase.*

6.3.1.2. Divisibilidad

Ejercicio 4.

1. ¿Qué números de Fibonacci son pares? ¿Cuáles son divisibles por 3? ¿Y por 5? ¿Qué números de Fibonacci son divisibles por 7?
2. ¿Qué podríamos decir acerca de los divisores de un número de Fibonacci que son también números de Fibonacci?
3. ¿Hay números de Fibonacci que sean primos?
4. ¿Es (experimentalmente) cierto que si un número de Fibonacci F_m es primo entonces m debe ser primo?
5. ¿Qué relación hay entre el MCD de dos números de Fibonacci, F_{m_1} y F_{m_2} , y el MCD de m_1 y m_2 ?

6. La sucesión de Fibonacci se puede definir, mediante la misma recursión, para las clases de restos módulo un entero m , es decir, en el anillo \mathbb{Z}_m . Como hay un número finito de clases de restos la sucesión debería repetirse a partir de un valor, es decir, debería ser periódica.

Estudia el período en función de m , es decir, el mínimo $k > 0$ tal que

$$F_k \equiv 0 \pmod{m}, F_{k+1} \equiv 1 \pmod{m},$$

y trata de encontrar algún orden en ese caos.

6.3.1.3. Curiosidad

Ejercicio 5. La suma de la serie

$$\sum_{k=2}^{\infty} \frac{F_k}{10^k}$$

es un número racional con un denominador menor que 100. Calcula el valor más razonable para esa fracción.

6.4. Algoritmo de Euclides

El algoritmo de Euclides para calcular el *máximo común divisor* de dos enteros es ciertamente el más antiguo que conocemos.

Dados dos enteros positivos a_0 y b_0 su máximo común divisor (MCD) es el mayor de sus divisores (positivos) comunes.

Obsérvese que hay un número finito de divisores (positivos) comunes, pues todos están comprendidos entre 1 y $\min(a, b)$.

El algoritmo de Euclides es un método para calcular el MCD. Supongamos dados dos enteros positivos, a_0 y b_0 , y queremos calcular su MCD.

1. Un primer paso será dividir el mayor, por ejemplo a_0 , entre el otro. Si el resto es 0 entonces a_0 es múltiplo de b_0 y el MCD es b_0 .
2. Si no, tendremos $a_0 = b_0 \cdot c_0 + r_0$ con $b_0 > r_0 > 0$. Esta última condición hace que se pueda dividir b_0 entre r_0 , para obtener $b_0 = r_0 \cdot c_1 + r_1$.

3. Si $r_1 = 0$ podemos sustituir b_0 en la primera división y obtenemos $a_0 = r_0 \cdot (c_1 \cdot c_0 + 1)$ de forma que b_0 y a_0 son múltiplos de r_0 . Falta comprobar que r_0 es el MCD.

Si no lo fuera, existiría un divisor común k mayor que r_0 . Tendremos entonces $k \cdot k' = c_1 \cdot r_0$ y $k \cdot k'' = (c_0 \cdot c_1 + 1) \cdot r_0$. Sustituyendo se obtiene $k \cdot k'' = c_0 \cdot k \cdot k' + r_0$ y finalmente $k(k'' - c_0 \cdot k') = r_0$ de forma que k no es el MCD.

4. Si $r_1 \neq 0$ hay que repetir el proceso, dividiendo r_0 entre r_1 , etc.
5. El algoritmo finaliza, después de un número finito de divisiones, porque los restos sucesivos son cada vez más pequeños pero siempre positivos o nulos. Cuando se alcance un resto nulo, el anterior es el MCD.
6. EJERCICIO: Demuestra esta última afirmación por inducción.

Una de las propiedades más útiles del MCD es el teorema de Bézout: *Si d es el MCD de a_0 y b_0 , entonces existen enteros u y v tales que $d = u \cdot a_0 + v \cdot b_0$.*

Se puede demostrar el teorema de Bézout como consecuencia del algoritmo de Euclides:

Supongamos, por ejemplo, que hemos hecho cuatro divisiones:

$$a_0 = b_0 \cdot c_0 + r_0$$

$$b_0 = r_0 \cdot c_1 + r_1$$

$$r_0 = r_1 \cdot c_2 + r_2$$

$$r_1 = r_2 \cdot c_3 + 0,$$

de forma que r_2 es el MCD de a_0 y b_0 ($a_0 \geq b_0$). Ahora podemos escribir, despejando los restos y sustituyendo,

$$r_2 = r_0 - r_1 \cdot c_2 = (a_0 - b_0 \cdot c_0) - (b_0 - (a_0 - b_0 \cdot c_0) \cdot c_1) \cdot c_2 = a_0(1 + c_1 \cdot c_2) + b_0(-c_0 - c_2 - c_0 \cdot c_1 \cdot c_2),$$

de forma que, en este caso en que hacemos 4 divisiones al aplicar el algoritmo, $u = 1 + c_1 \cdot c_2$ y $v = -c_0 - c_2 - c_0 \cdot c_1 \cdot c_2$.

Debe ser claro que si necesitamos más de cuatro divisiones para llegar al MCD, el procedimiento para calcular los coeficientes u y v en el teorema de Bézout va a funcionar como en el caso de cuatro.

6.4.1. Mínimo común múltiplo

Dados dos enteros positivos, a_0 y b_0 , su *mínimo común múltiplo* (MCM) es el menor de sus múltiplos comunes. Los dos enteros tienen una infinidad de múltiplos comunes, por lo menos todos los enteros de la forma $k \cdot a_0 \cdot b_0$, pero todo conjunto no vacío de enteros positivos tiene un menor elemento, y, por tanto, el MCM está bien definido.

Una manera fácil de calcular el MCM de dos enteros es, debido a que podemos calcular su MCD mediante el algoritmo de Euclides, usando el hecho de que el producto de dos enteros positivos cualesquiera es igual al producto de su MCD por su MCM.

6.4.2. MCD en Sage

En Sage disponemos de dos instrucciones principales relacionadas con el MCD: `gcd(a,b)` y `xgcd(a,b)`. La primera devuelve el MCD de los dos enteros y la segunda una 3-upla (d, u, v) en la que el primer elemento es el MCD de a y b , y los otros dos son enteros tales que $d = u \cdot a + v \cdot b$, es decir, es decir se calculan un par de enteros tales que se verifica el teorema de Bézout.

Los enteros u y v no son únicos, de hecho, la ecuación $ax + by = d$, con d el MCD de a y b , tiene soluciones

$$x = u + (b/d)k, \quad y = v - (a/d)k,$$

para algún k entero, con u y v una solución particular, por ejemplo la dada por el algoritmo de Euclides.

Estas dos funciones de Sage también pueden aplicarse a polinomios en una variable con coeficientes racionales, ya que tales polinomios admiten división con resto y, en consecuencia, el algoritmo de Euclides es válido para ellos.

Ejercicio 6.

1. Programa el algoritmo de Euclides en forma iterativa y en forma recursiva.
2. Programa, en forma iterativa y en forma recursiva, el algoritmo, explicado más arriba, que nos da los coeficientes u y v en el teorema de Bézout.
3. El algoritmo iterativo para calcular los coeficientes que aparecen en el teorema de Bézout se puede organizar mucho mejor usando multiplicación de matrices. Esto es, hasta cierto punto, análogo al cálculo de los términos de la sucesión de Fibonacci usando también multiplicación de matrices.

Programar el algoritmo de esta manera.

4. Llamamos ecuación diofántica lineal con dos incógnitas a una de la forma

$$a \cdot x + b \cdot y = c,$$

donde a, b, c son enteros dados, y x e y , las soluciones, deben ser también enteros tales que se cumple la ecuación.

El teorema de Bézout permite resolver esta clase de ecuaciones. Escribe código tal que dados los coeficientes a, b y c devuelva las infinitas soluciones de la ecuación diofántica como una fórmula en función de un parámetro k entero.

5. Escribe una función de Sage tal que dados enteros a y b devuelva el número de divisiones que hay que hacer al aplicar el algoritmo de Euclides.

Queremos estudiar, experimentalmente, la siguiente cuestión: “Encontrar, en función de k , el número máximo de divisiones que hay que hacer para calcular el MCD de un número a de más de k cifras y un número b de exactamente k cifras”.

6.5. Números primos

Un número $p > 1$ es primo si sus únicos divisores son ± 1 y $\pm p$ ². Uno de los teoremas fundamentales, ya conocido por Euclides, afirma que *todo entero ≥ 2 es el producto, único salvo el orden, de un número finito de enteros primos*.

También Euclides sabía que hay infinitos números primos, con el siguiente argumento:

“Si hubiera un número finito de primos p_1, p_2, \dots, p_n el entero $p_1 \cdot p_2 \cdot \dots \cdot p_n + 1$ no sería divisible por ninguno de ellos luego sería, gracias al teorema de factorización, necesariamente primo.”

Si observamos la sucesión de los primos inmediatamente vemos que según avanza los huecos entre primos se hacen, *en general*, cada vez más grandes. Dicho de otra manera, podemos generar listas tan grandes como queramos de enteros consecutivos todos compuestos:

$$n! + 2, n! + 3, n! + 4, \dots, n! + n.$$

²Esta es la definición que se usa habitualmente en la enseñanza preuniversitaria, pero no es del todo correcta. Lo definido por ella es lo que llamamos **elemento irreducible** en un anillo, mientras que la definición correcta de **primo** afirma que *un elemento de un anillo es primo si cada vez que divide a un producto de dos elementos divide a uno de los factores*. En el anillo de los enteros las dos nociones coinciden, pero no es cierto en general que los elementos primos sean los mismos que los irreducibles en todo anillo.

Entonces, uno de los problemas centrales, en la teoría de los números primos, es el de *la distribución de los números primos dentro del conjunto de los enteros*, que podemos entender como la determinación del número, aproximado, de primos en el intervalo $[N_1, N_2]$.

Si definimos $\pi(x)$ como el número de primos menores o iguales que el número real x el *teorema de los números primos*, conjeturado por Legendre y Gauss, y demostrado por Hadamard y de la Vallée-Poussin, propone que

$$\lim_{n \rightarrow \infty} \frac{\pi(x)}{x/\ln(x)} = 1.$$

Este resultado, y otros muchos en teoría de números, se demostró usando técnicas de cálculo diferencial para funciones de una variable compleja, y todavía es la demostración más sencilla aunque hay otras.

El teorema nos informa, con una aproximación bastante buena, de que el número de primos $\leq x$ es $x/\ln(x)$.

Una de sus consecuencias es que el primo n -ésimo p_n está próximo al real $n \cdot \ln(n)$, aunque es siempre mayor (Rosser, 1938).

6.5.1. Números primos en Sage

Algunas de las funciones o métodos disponibles en Sage para tratar con números primos:

1. En primer lugar tenemos el método `.is_prime()`, que devuelve **True** o **False** según el entero, de Sage, al que se aplique sea primo o no.
2. El método `.next_prime()` devuelve el primo siguiente al entero al que se aplica. Sirve, por ejemplo, para localizar sucesiones grandes de enteros consecutivos que sean todos compuestos.
3. La función `prime_range(N_1, N_2)` devuelve la lista de los primos contenidos en el intervalo $[N_1, N_2]$.
4. La función `nth_prime(m)` devuelve el primo, p_m , que ocupa el lugar m -ésimo en la sucesión de los números primos.
5. La función `primes(N_1, N_2)` es un iterador sobre la lista de los primos. Sirve para definir un bucle que itere sobre los enteros primos en el rango $[N_1, N_2]$ utilizando poca memoria RAM.
6. Ya discutimos, en la página 63, cómo trabajar con las factorizaciones de un entero como producto de primos.

6.5.2. Algunos problemas

Dentro de la teoría de los números primos hay bastantes conjeturas, resultados probablemente ciertos pero para los que no se conoce demostración, y resultados, algunos de demostración extremadamente difícil, pero en muchos casos con un enunciado muy simple.

A continuación hay una pequeña muestra, y para los problemas marcados con un asterisco SE PROPONE QUE SE ELABOREN PROGRAMAS para estudiarlos.

1. CONJETURA DE GOLDBACH:

- a) (Goldbach fuerte *) Todo número par, mayor que 2, es la suma de dos números primos.
- b) (Goldbach débil *) Todo número impar mayor que cinco es la suma de tres números primos. Esta conjetura parece que ha sido finalmente probada.

Puedes leer el comienzo de este [artículo](#), en el que el autor de la demostración, el matemático peruano Harald Andrés Helfgott, expone las ideas básicas. Lo que nos interesa aquí es que la demostración tiene dos partes:

- 1) Para números impares menores que 10^{29} se comprobó mediante ordenador.
- 2) Para los mayores que 10^{29} Helfgott mejoró argumentos conocidos hasta llegar a demostrar la conjetura para todos ellos.
- c) También se ha probado (Chen, 1973) que todo número par *suficientemente grande* es la suma de dos primos o bien de un primo y un producto de dos primos (por definición un *semiprimo*).
- d) (Conjetura de Lemoine) Todo número impar mayor que cinco es la suma de un número primo y de un semiprimo par.
- e) (Conjetura de Sun *) Todo número impar mayor que tres es la suma de un número primo impar y un producto de dos enteros consecutivos.

2. PRIMOS GEMELOS:

- a) (*) Existen infinitos pares de números primos de la forma $(p, p + 2)$.

- b) Al comienzo de 2013 se ha presentado (Zhang) una demostración de que hay infinitos pares de primos que distan menos que 70 millones. Esto está bastante lejos de la conjetura, pero en este momento parece que la cota, 70 millones, ya ha sido rebajada a un número del orden de 4000.
- c) (*) Se sabe que todos los pares de primos gemelos $(p, p + 2)$, $p \geq 5$ son de la forma $(6n - 1, 6n + 1)$ para algún n . Esto se debe a que todo primo mayor o igual a cinco es de la forma $6n - 1$ o de la forma $6n + 1$.
- d) (*) EJERCICIO: encontrar todos los enteros p tales que $p, p + 4, p + 6, p + 10, p + 12, p + 16, p + 22$ sean todos primos.

3. PRIMOS COMO VALORES DE POLINOMIOS :

- a) (*) Dirichlet demostró que los polinomios $an + b$, con $MCD(a, b) = 1$, toman infinitos valores primos, es decir, *hay infinitos primos en las progresiones aritméticas en las que no todos los elementos son múltiplos de un entero d* .
Por ejemplo, es un ejercicio fácil, usando la misma idea que la demostración de Euclides de la existencia de infinitos primos, demostrar que hay infinitos primos en la progresión aritmética $4n + 3$, pero el caso general es bastante difícil.
- b) (*) No se sabe si hay infinitos primos de la forma $n^2 + 1$. Sin embargo se sabe (Iwaniec, 1978) que hay infinitos enteros de la forma $n^2 + 1$ que tienen a lo más dos factores primos.
- c) No se conoce ningún polinomio de una variable, con coeficientes enteros y de grado mayor que uno, que tome infinitos valores primos.
- d) El polinomio $n^2 + n + 41$ comienza bien, toma valores primos para n entre 0 y 39, pero no se sabe que tome infinitos valores primos.
- e) CONJETURA DE BOUNIAKOVSKY (*): Sea $P(x)$ un polinomio con coeficientes enteros irreducible (i.e. no se puede escribir como producto de dos polinomios con coeficientes enteros) y de grado al menos 1. Sea d el mayor entero tal que divide a todos los valores $P(n)$, $n \in \mathbb{Z}$. Si d vale 1 entonces $P(n)$ tiene infinitos valores primos.
- f) (*) Sin embargo, hay polinomios de dos variables como $x^2 + y^2$ para los que se puede demostrar, usando el teorema de Dirichlet, que toman infinitos valores primos.
- g) Hay polinomios con coeficientes enteros en 10 variables tales que el conjunto de los valores positivos que toman, cuando las variables recorren los enteros no negativos, es el conjunto de enteros primos.

6.6. Enteros representables como sumas de cuadrados

La suma de los cuadrados de dos enteros, $x^2 + y^2$, es un entero no negativo y queremos averiguar qué enteros no negativos se pueden representar como una tal suma de cuadrados. Es claro que, por ejemplo, $2 = 1^2 + 1^2$ pero 3 no se puede representar como una suma de cuadrados. En estos ejercicios se trata de generar un número suficiente de ejemplos de enteros que se representan, o no, como sumas de cuadrados, y de obtener conjeturas razonables acerca de la condición (condiciones) que debe verificar un entero n para que se pueda representar como una suma de cuadrados.

Ejercicio 7.

1. ¿Qué números primos se pueden representar como suma de dos cuadrados? Mediante un programa adecuado no es difícil ver cuál debe ser la respuesta, aunque no es tan fácil demostrarla.
2. ¿Qué enteros positivos se pueden representar como una suma de cuadrados?

Gracias a las identidades

$$\begin{aligned}(x_1^2 + y_1^2)(x_2^2 + y_2^2) &= (x_1x_2 \pm y_1y_2)^2 + (x_1y_2 \mp y_1x_2)^2 \\ 2(x_1^2 + y_1^2) &= (x_1 + y_1)^2 + (x_1 - y_1)^2\end{aligned}$$

no es difícil responder a esta pregunta usando la respuesta de la anterior y el hecho de que todo entero es un producto, esencialmente único, de potencias de primos.

3. ¿Qué enteros positivos se pueden representar como una suma de tres cuadrados? ¿Y como una suma de cuatro cuadrados?

6.7. Desarrollos decimales

Dada una fracción reducida n/d , con n y d enteros positivos primos entre sí, podemos asociarle un número real, decimal con infinitas cifras, mediante el algoritmo de división con resto de números enteros.

1. Para empezar, si $n > d$, separamos la parte entera dividiendo $n = c \cdot d + r$, de forma que $\frac{n}{d} = c + \frac{r}{d}$ con c la parte entera de la fracción. Podemos continuar con el desarrollo decimal de r/d , es decir, suponiendo que $n < d$.

2. Si el denominador d es de la forma $2^a \cdot 5^b$, llamemos n al máximo de a y b , obtenemos un número entero multiplicando r/d por 10^n , y, por tanto el desarrollo decimal de r/d es cero a partir del lugar n -ésimo después del punto decimal.
3. Supongamos ahora que $n < d$ y $MCD(d, 10) = 1$. ¿Cómo podemos calcular la primera cifra decimal de n/d ? Es claro que la primera cifra decimal es la parte entera de $(10 \cdot n)/d$.

Por ejemplo, la primera cifra decimal de $3/7$ es 4 como vemos al realizar la división entera de $3 \cdot 10 = 30$ por 7, esto es $30 = 4 \cdot 7 + 2$.

¿Y la segunda cifra decimal? Habrá que restar a $(10 \cdot n)/d$ su parte entera, multiplicar por 10 otra vez y quedarnos con la parte entera del resultado. En el ejemplo, y dividiendo la división entera ya realizada por 7, encontramos $\frac{30}{7} - 4 = \frac{2}{7}$, y continuaríamos con el desarrollo decimal para $2/7$.

4. Defino entonces dos funciones

- a) $f(r) := \text{floor}(10 \cdot r)$, que aplicada a un número racional menor que 1 produce la primera cifra de su desarrollo decimal.
- b) $F(r) := 10 \cdot r - \text{floor}(10 \cdot r)$, que aplicada a un número racional $r = n/d$ produce otro número racional $r_1 := n_1/d < 1$, con el mismo denominador que r , que es la fracción que contiene la información sobre las siguientes cifras decimales de n/d . En particular, aplicando f al racional $F(r)$ obtenemos la segunda cifra decimal de r .

5. Como todas las fracciones que obtenemos al iterar F tienen el mismo denominador d , sólo puede haber un número finito de numeradores, enteros entre 1 y $d - 1$, lo que sugiere que tiene sentido considerar la cuestión en el anillo de clases de restos módulo d .
6. ¿Quién es el numerador n_1 de la fracción r_1 ? Si escribimos $10 \cdot n = c \cdot d + r$ y dividimos entre d ambos miembros vemos que n_1 es el resto r de esta división, de forma que cuando consideramos la función F como actuando en los numeradores de fracciones que tienen d como denominador, se trata simplemente de la función, de \mathbb{Z}_d en sí mismo, definida mediante $\overline{F}([n]) := [10 \cdot n]$, y sabemos, gracias al teorema de Fermat-Euler (ver la subsección 6.2.1), que existe un exponente ν tal que $\overline{F}^\nu = \text{Identidad}$.

Esto demuestra que *los desarrollos decimales de fracciones n/d con $n < d$ y $MCD(d, 10) = 1$ son periódicos.*

7. Por último, si el denominador d de la fracción n/d que estamos estudiando es de la forma $2^a \cdot 5^b \cdot D$ con $MCD(D, 10) = 1$, y multiplicamos la fracción por $10^{\max(a,b)}$ obtenemos una expresión de la forma $C + \frac{N}{D}$ con C un entero. Ya sabemos que

el desarrollo decimal de N/D es periódico, de manera que dividiendo por $10^{\max(a,b)}$ obtenemos que el desarrollo decimal de la fracción original tiene un primer bloque de la forma $C/10^{\max(a,b)}$ y a continuación otro bloque repetido infinitas veces, al que llamamos la *parte periódica del desarrollo*.

¿Cómo podemos obtener la fracción que corresponde a un cierto desarrollo decimal periódico? Supongamos, por ejemplo, un desarrollo de la forma

$$x := 2.34545454545 \dots =: 2.3\overline{45}.$$

Podemos escribir $x = \frac{23}{10} + \frac{0.\overline{45}}{10}$. Tomando $s = 0.\overline{45}$, y restádoselo a $100s = 45.\overline{45}$ se obtiene $s = \frac{45}{99} = \frac{5}{11}$ y así

$$x = \frac{23}{10} + \frac{\frac{5}{11}}{10} = \dots = \frac{129}{55}$$

Ejercicio 8.

1. Programa una función, o varias, para calcular el desarrollo decimal de una fracción n/d reducida cualquiera.
2. Programa una función para calcular la fracción que corresponde a un desarrollo decimal dado.
3. ¿Qué modificaciones habría que hacer para que estos programas funcionaran para fracciones de enteros expresados en el *sistema de numeración de base b* ?

6.8. Ejercicios

Ejercicio 9. La congruencia de Wilson afirma que para todo número primo p se verifica que el factorial de $p-1$ es congruente con -1 módulo p . Es interesante que el recíproco también es cierto: si un entero n verifica la congruencia de Wilson entonces es primo, y por tanto la congruencia de Wilson puede servir como un criterio de primalidad.

El método más simple, pero no el más eficiente, para determinar si un número n es primo es la criba de Eratóstenes, es decir, n es primo si después de cribar la lista de enteros entre 2 y n , ambos inclusive, n está todavía en la lista cribada. Se reproduce un código de criba a continuación:

```

def criba(n):
    '''n al menos 2'''
    aux = [True]*int(n)
    aux[0] = False
    aux[1] = False
    for i in xrange(2,floor(sqrt(n))+1):
        if aux[i]:
            for j in xrange(i*i,n,i):
                aux[j] = False
    return [k for k in xrange(n) if aux[k]==True]

```

En este ejercicio debes comparar, en términos de eficiencia, la criba con el criterio de Wilson como método para decidir si un entero n es primo, es decir, debes producir evidencia que demuestre, suficientemente, cuál de los dos métodos es mejor.

Ejercicio 10. En este ejercicio estudiamos diversos métodos para calcular el número binomial $\binom{n}{m}$, que por definición es el entero

$$\binom{n}{m} := \frac{n!}{m!(n-m)!},$$

y cuenta el número de subconjuntos con m elementos de un conjunto de n elementos.

1. Un primer método, que usaremos como base para nuestras comparaciones, consiste en usar la definición anterior directamente, es decir, calculando los factoriales y realizando las operaciones indicadas. Define una función de SAGE de nombre `binomial_1(n,m)` y que devuelva el binomial calculado de esta manera.
2. El cálculo anterior se puede organizar mejor: simplificamos $(n-m)!$ con parte de $n!$ y utilizamos que queda el mismo número de factores en el numerador que en el denominador para calcular el binomial más eficientemente. Define una función de SAGE de nombre `binomial_2(n,m)` y que devuelva el factorial calculado de esta manera.
3. Otra manera de calcular el binomial es la base del **triángulo de Tartaglia**, es decir, las relaciones

$$\binom{n}{0} = \binom{n}{n} = 1; \quad \binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m},$$

que permitirían una definición recursiva del binomial. NO se pide esta solución recursiva, que no sería nada eficiente, sino una iterativa equivalente. Define una función de Sage de nombre *binomial_3*(n, m) que utilice bucles y las relaciones anteriores para calcular el binomial. En cierto sentido este programa haría lo mismo que cuando calculamos el triángulo de Tartaglia a mano. Debes tener especial cuidado con el uso de la RAM.

4. Compara la eficiencia de los tres programas, tratando de decidir, de forma concluyente, cuál es mejor.

Ejercicio 11.

Estudiamos una manera, que esperamos sea eficiente, de calcular factoriales basada en el siguiente teorema:

Dados un entero n y un primo $p \leq n$ el exponente de p en la factorización de $n!$ en producto de primos es el entero

$$\text{exponente}(p) := \sum_{i=1}^{i=\infty} \text{floor}(n/p^i).$$

Observa que la suma es en realidad finita, ya que en cuanto p^i supera a n los sumandos son todos cero, y para todos los primos $p \leq n$ el exponente correspondiente es al menos 1.

1. Define una función que reciba el entero n como argumento y devuelva una lista de pares con el primer elemento de cada par uno de los primos $p \leq n$ y el segundo el exponente que corresponde a p de acuerdo al teorema mencionado.
2. Define una función de nombre *factorial_(n)* que en primer lugar llame a la función del apartado anterior, y luego calcule el producto de todos los primos elevado cada uno a su exponente. El resultado debe ser el factorial de n . Para elevar cada primo a su exponente debe usarse el algoritmo eficiente visto en **el capítulo anterior**.
3. Compara sistemáticamente los tiempos de ejecución del factorial de Sage y del tuyo. ¿Qué conclusión sacas?

Ejercicio 12. Decimos que un entero positivo n es **multiplicativamente perfecto** si el producto de todos los divisores de n vale exactamente n^2 . El ejemplo más sencillo de un número multiplicativamente perfecto es el producto $n = p \cdot q$ de dos primos distintos y, por tanto, existen infinitos enteros multiplicativamente perfectos. Queremos caracterizarlos.

1. Primero define una función de Sage, de nombre *perfecto*(n), que reciba como argumento un entero n y devuelva **True** o **False** según el número n sea multiplicativamente perfecto o no.

2. Ahora define una función de Sage, de nombre *perfectos*(N), que reciba como argumento un entero N y devuelva la lista de todos los enteros multiplicativamente perfectos que pertenecen al intervalo $[1, N]$.
3. Usando las listas de enteros multiplicativamente perfectos que puedes obtener con la función del apartado anterior, produce una conjetura razonable acerca de qué enteros son multiplicativamente perfectos.
4. Define una tercera función, de nombre *comprobar*(N), que devuelva **True** si tu conjetura es correcta para enteros positivos menores que N , y **False** si no lo es.
5. Experimenta con valores de N suficientemente grandes para obtener el $N = 10^t$ (t entero) más grande tal que tu programa *comprobar*(N) se ejecuta en menos de un minuto.

Ejercicio 13. En este ejercicio estudiamos la existencia de raíces k -ésimas en el anillo \mathbb{Z}_m de clases de restos módulo un entero m . Como es natural, dados elementos $a, b \in \mathbb{Z}_m$, decimos que a es una raíz k -ésima de b si $a^k = b$ en \mathbb{Z}_m . Nuestro objetivo es, dado el entero m , determinar los valores de $2 \leq k \leq m$ tales que todos los elementos de \mathbb{Z}_m tienen una raíz k -ésima.

1. Define una función de Sage, con nombre *raices*(m, k), que devuelva **True** si todas las clases de restos módulo m tienen una raíz k -ésima, con $2 \leq k \leq m$, y **False** en caso contrario.
2. Aplica la función *raices*(m, k) con m primo, por ejemplo $m = 23$, y trata de entender (explicita una conjetura) cuáles son los valores de k para los que se obtiene **True**.
3. Define una nueva función de Sage, *comprobador*(N), que sirva para comprobar si tu conjetura es cierta para todos los enteros primos del intervalo $[1, N]$. Mediante esta función comprueba tu conjetura hasta el $N = 10^t$ (t entero) más grande tal que la comprobación tarde menos de un minuto.
4. Si todavía tienes ganas puedes estudiar el caso de enteros m que son producto de dos primos distintos.

Ejercicio 14.

Capítulo 7

Aproximación

Sage, ya que es un manipulador simbólico, es capaz de manejar símbolos como $\sqrt{2}$, e o π , sin “cometer ningún error de aproximación”. Pero en ocasiones deseamos tener un valor aproximado (una expresión decimal finita) de las cantidades simbólicas que manejamos. Esto se puede conseguir de forma sencilla por medio de la función **numerical_approx**, o de sus alias **n** y **N** (las tres funciones se pueden utilizar también como métodos). Así, por ejemplo,

```
n(pi)  
3.14159265358979
```

da una aproximación de π con 53 bits (dígitos binarios) de precisión. Si deseamos más o menos precisión, podemos especificarla por medio de las opciones `prec`, que indica el número de dígitos binarios de precisión, y `digits`, que permite elegir el número de dígitos decimales de precisión. De esta forma,

```
a=numerical_approx(sqrt(2),prec=4)  
a; a.str(base=2)  
1.4  
'1.011'
```

mientras que

```
b=N(sqrt(2),digits=4)
b; b.str(base=2)
```

```
1.414
'1.0110101000001010'
```

Las aproximaciones numéricas de números reales se almacenan internamente en base 2. Por consiguiente, dos números cuya expansión decimal parece ser la misma pueden ser diferentes:

```
x=N(pi,digits=3)
y=N(3.14,digits=3)
x; y; x==y; x.str(base=2); y.str(base=2)
```

```
3.14
3.14
False
'11.001001000100'
'11.001000111101'
```

Esta situación, que es ciertamente desagradable, es inherente al cálculo con números decimales, y lo que se intenta, en *Cálculo Numérico*, es controlar los errores que inevitablemente se producen.

En este capítulo se incluyen diversos ejemplos alrededor de la idea de *aproximación*:

1. Métodos, usando fórmulas recursivas o series, para obtener aproximaciones a π . En particular, una serie que permite calcular la cifra de π que ocupa el lugar n -ésimo sin calcular las cifras anteriores.
2. Fracciones continuas como medio de aproximar números irracionales mediante racionales.
3. Veremos un truco, usando logaritmos, para obtener la cifra dominante, la primera por la izquierda, de enteros de la forma a^b .
4. Aproximación de los ceros de funciones definidas en un intervalo de \mathbb{R} .
5. Aproximaciones locales, serie de Taylor, y globales a funciones de una variable real.

7.1. Precisión

En Sage es fácil aumentar la precisión con la que efectuamos los cálculos con números decimales.

Una instrucción como `NR = RealField(prec=n)` define números reales de n bits, ocupan n bits en la memoria del ordenador. Por defecto, si se usa `RR` como cuerpo de reales la precisión es de 53 bits.

Forzamos un número decimal, por ejemplo π , a estar en `NR`, y, por tanto, a intervenir en los cálculos como un decimal de n bits mediante `NR(pi)`. El número de cifras decimales con las que aparece el resultado viene a ser del orden de $n/4$, es decir cada cifra ocupa unos cuatro bits.

Como ya **sabemos**, otra manera de forzar la evaluación como decimal de un número, por ejemplo π , es mediante `N(pi,prec=n)`, con la variante `N(pi,digits=m)` que devuelve m dígitos decimales de π .

¿Para qué queremos aumentar la precisión de un cálculo? Cualquier cálculo con números decimales puede tener *errores de redondeo*, lo que hace que algunas de las últimas cifras devueltas por el cálculo pueden ser incorrectas. Incrementando la precisión suficientemente podemos comprobar si esas cifras dudosas se mantienen o cambian. Desde un punto de vista práctico, y dentro de este curso, que no es uno de Cálculo Numérico, podemos decir que las cifras del resultado de un cálculo que se mantienen al incrementar la precisión *deberían ser correctas*.

En la hoja de Sage [71-APROX-ejemplos.ipynb](#) se muestran algunos ejemplos¹ de los errores que se pueden producir al tratar con aproximaciones de números reales.

7.2. El número π

Como se verá a lo largo de esta sección el cálculo de aproximaciones al valor de la constante π tiene una larga historia. Por ejemplo, en la Biblia se afirma que el valor de π es 3 con lo que se comete un error relativo de (aprox) $(3.141592 - 3)/3.141592 = 0.045070$, es decir, (aprox) un 4.5 %. También se usaban en la antigüedad fracciones que como $256/81 = 3.16 \dots$ o $22/7 = 3.1428 \dots$ aproximaban mejor o peor el valor de π .

Arquímedes ya entendió que había que producir un algoritmo que suministrara tantas cifras correctas de π como quisiéramos, y su algoritmo iterativo hace exactamente eso. Con la llegada del cálculo diferencial aparecieron métodos basados en series o productos infinitos que convergen a π a mayor o menor *velocidad*. La mayoría de los ejemplos que estudiaremos son series.

Una buena serie debe producir muchos dígitos nuevos de π con cada sumando que añadimos (gran velocidad de convergencia), pero al mismo tiempo el cálculo de cada uno de esos sumandos no debe ser muy costoso.

¹Tomados esencialmente del libro de N.J Highman *Accuracy and stability of numerical algorithms*.

¿Es importante conocer cuatrillones de cifras de π ?

1. La constante π aparece en innumerables fórmulas en matemáticas y física, y en consecuencia también en ingeniería, pero para su uso en ciencia e ingeniería casi siempre basta con unas pocas cifras decimales, digamos 8 o 16.
2. Sin embargo estos cálculos tienen alguna aplicación práctica:
 - a) Se usan para comprobar que *hardware* nuevo funciona correctamente. En el caso de *hardware* con defectos de diseño, sobre todo en la parte del sistema que se ocupa de las operaciones aritméticas, pueden aparecer errores sistemáticos en los dígitos de π calculados.
 - b) Se han utilizado enormes superordenadores para calcular dígitos de π , creo que en los 90 pero ya no, y cabe pensar que estos cálculos han tenido alguna influencia en la puesta a punto de los superordenadores.
 - c) Actualmente se intenta calcular trillones de cifras de π usando ordenadores de sobremesa convenientemente adaptados. Aparecen problemas interesantes de almacenamiento del resultado, hacen falta varios discos duros de varios TB cada uno, y otros debidos a que el programa debe funcionar durante varios meses y cualquier error que se produzca puede estropear el resultado.
 - d) En criptografía hay al menos un algoritmo, el llamado “Blowfish”, que utiliza las primeras 8366 cifras hexadecimales de π .
3. Aunque no parece que sea un asunto central en las Matemáticas, el descubrimiento de nuevas series que convergen a π , o a otra de las constantes importantes, tiene cierto interés, sobre todo si las series convergen rápido.

Dado que, aunque la idea de π es importante, conocer en detalle sus cifras no es tan tremendamente útil, podemos preguntarnos ¿por qué es *tan popular*?

1. Un primer motivo es que todos lo conocemos desde la Primaria, memorizamos algunas cifras, y ahí queda junto a la fórmula para resolver la ecuación de segundo grado y alguna cosa más.
2. Hay ciertas competiciones asociadas al cálculo de cifras de π : récord en el número de cifras calculadas usando *hardware* arbitrario, récord en el número de cifras calculadas usando ordenadores de “sobremesa”, récord en la memorización de cifras consecutivas de π , o [estos](#).

Algunos de estos logros aparecen en la prensa y producen un cierto revuelo. Por ejemplo, es bastante interesante este [artículo](#) aparecido en *The New Yorker* en el que se cuenta algo de la relación de los hermanos Chudnovsky con el cálculo de cifras de π .

3. Hay una [película](#), π (1998), dirigida por Darren Aronofsky en la que las cifras de π juegan un gran papel. De hecho, pienso que la película pudo estar en gran parte influida por la lectura del artículo sobre los hermanos Chudnovsky (1992). El argumento viene a ser “un matemático loco, o enloquecido, busca la respuesta a todos *los secretos del Universo* en las cifras de π ”.
4. En la película aparecen referencias a sectas judías, más o menos secretas, que asignan cierto valor místico al número π . Por ejemplo, puede ser interesante hojear este [artículo](#), en el que se relaciona el número π con el estudio cabalístico de la Torah.
5. Desde hace algún tiempo se celebra en colegios y universidades el *Pi day* que, naturalmente, cae en el 14 de Marzo, es decir, el 03/14 en la forma norteamericana de representar las fechas. Las celebraciones incluyen el consumo de tartas, *Pi day* suena igual que *Pie day* (el día de las tartas), recitación de largas series de dígitos de π , etc.
6. Páginas web, como [esta](#), permiten buscar subcadenas dentro de las primeras dos mil millones de cifras decimales de π . Puedo entonces determinar que mi DNI aparece 16 veces y mi fecha de nacimiento 18 veces. ¿Hay algún motivo para que los dos números aparezcan un número similar de veces? Volveremos sobre este asunto.
7. Otras razones que ahora no se me ocurren.

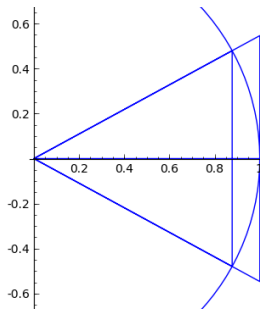
Es posible encontrar en la red bastantes artículos relacionados con el cálculo de π . Dos muy recomendables son:

1. [Una historia](#) del cálculo de las cifras de π escrito por cuatro de los mayores expertos.
2. [Un artículo](#) sobre la importancia del número π en las Matemáticas. Recoge una conferencia impartida durante el *Pi day* de 2011 en la Universidad de Utah.

7.2.1. Arquímedes (S. III a.C.)

Con lo que sabemos sobre las funciones trigonométricas, podemos razonar fácilmente que el perímetro p_n de un polígono regular de n lados inscrito en la circunferencia unidad, y el perímetro P_n del circunscrito regular de n lados, serán:

$$p_n = 2n \operatorname{sen}(\pi/n) < 2\pi < P_n = 2n \tan(\pi/n).$$



Pero Arquímedes no disponía de esas funciones trigonométricas, ni de una calculadora que le diese el valor de π , de forma que lo que hizo fué construir un procedimiento recursivo que relacionaba los perímetros de un polígono regular inscrito de n lados y el de $2n$ lados.

Ejercicio 1.

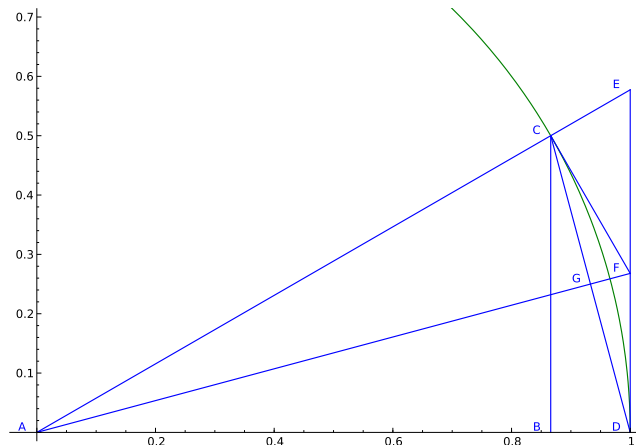
1. Utilizar la sucesión p_n para calcular aproximaciones a 2π . Como debemos usar un valor aproximado de π para poder calcular p_n este método es una tontería. Veremos métodos mejores.
2. ¿Por qué es convergente la sucesión p_n ? Contestar requiere un argumento matemático.
3. ¿Cómo podemos ver en las aproximaciones numéricas que la sucesión es de Cauchy?

Empezamos con dos hexágonos, que tienen

Llamemos 2α al ángulo que vemos en el origen en la siguiente figura, desde el eje OX hasta el segundo radio azul, y supongamos que abarcase uno de los lados del polígono de $2n$ lados *inscrito* en el círculo (la cuerda CD que vemos en la figura). Llamaremos:

1. $S_{2n} := \ell(CG)$ es la mitad de la longitud del lado del polígono regular de $2n$ lados inscrito en la circunferencia.
2. $S_n := \ell(CB)$ es la mitad de la longitud del lado del polígono regular de n lados inscrito en la circunferencia.
3. $T_{2n} := \ell(CF)$ es la mitad de la longitud del lado del polígono regular de $2n$ lados circunscrito a la circunferencia.
4. $T_n := \ell(ED)$ es la mitad de la longitud del lado del polígono regular de n lados circunscrito a la circunferencia.

Así que los perímetros con n lados son: $p_n = 2nS_n$, $P_n = 2nT_n$.



Debe ser claro que:

1. Los triángulos $\triangle ABC$, $\triangle ADE$ y $\triangle EFC$ son los tres semejantes entre sí.
2. Los triángulos $\triangle ADF$, $\triangle CGF$ y $\triangle CBD$ son también semejantes entre sí.
3. Los triángulos $\triangle ACF$ y $\triangle ADF$ son iguales.

Como consecuencia se obtienen las siguientes relaciones:

1. (Usando 1 y 3)

$$\frac{BC}{DE} = \cos(2\alpha) = \cos(\widehat{AFE}) = \frac{CF}{FE},$$

es decir,

$$\frac{S_n}{T_n} = \frac{T_{2n}}{T_n - T_{2n}},$$

y operando

$$\frac{T_n}{S_n} = \frac{T_n - T_{2n}}{T_{2n}} = \frac{T_n}{T_{2n}} - 1$$

y finalmente

$$T_{2n} = \frac{1}{\frac{1}{T_n} + \frac{1}{S_n}}.$$

2. (Usando 2)

$$\frac{\ell(CB)}{\ell(CD)} = \cos(\alpha) = \frac{\ell(CG)}{\ell(CF)},$$

y, por tanto,

$$\frac{S_n}{2S_{2n}} = \frac{S_{2n}}{T_{2n}},$$

de donde

$$S_{2n} = \sqrt{\frac{T_{2n}S_n}{2}}.$$

Estas fórmulas permiten entonces calcular (S_{2n}, T_{2n}) supuesto que conocemos (S_n, T_n) , y partiendo de $(S_6, T_6) = (1/2, 1/\sqrt{3})$ como caso inicial podemos calcular los perímetros de los polígonos regulares inscrito y circunscrito con número de lados de la forma $6 \cdot 2^k$, y así aproximar la longitud de la circunferencia 2π como

$$\lim_{k \rightarrow \infty} 2(6 \cdot 2^k)S_{6 \cdot 2^k} = \lim_{k \rightarrow \infty} 2(6 \cdot 2^k)T_{6 \cdot 2^k}.$$

Ejercicio 2.

Implementa en Sage este algoritmo de Arquímedes para aproximar π , calculando hasta que la diferencia entre el perímetro del polígono circunscrito y el del inscrito sea menor que 10^{-n_0} para un cierto n_0 fijado a priori.

7.2.2. Leonhard Euler (1707-1783)

Con lo mucho que se había aprendido desde Arquímedes, Euler ya sabía que $\arctan(x)' = 1/(1+x^2)$, y que para $|x| < 1$, podemos ver esa fracción como la suma de una progresión geométrica de razón $-x^2$ y tratar de “integrar esa suma infinita término a término como si fuese un polinomio”:

$$\frac{1}{1+x^2} = 1 - x^2 + x^4 - x^6 + x^8 - \dots$$

que nos da, integrando término a término,

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \dots \quad (7.1)$$

Como $\tan(\pi/4) = 1$, eso da una manera de aproximar $\pi/4$, sumando para $x = 1$, pero muy lenta y, por tanto, muy poco eficiente.

La ingeniosa idea de Euler fue que, en vista de que $\tan(a+b) = (\tan(a) + \tan(b))/(1 - \tan(a)\tan(b))$, los ángulos $a = \arctan(1/2)$, $b = \arctan(1/3)$ suman $\pi/4$, y la suma (7.1) para esos valores de x , más próximos a 0, converge mucho más rápido.

Ejercicio 3.

Implementa en Sage ambas formas de aproximar π (sumando (7.1) para $x = 1$ y sumando los valores que se obtienen para $x = a$ y $x = b$) y compara la eficiencia de los dos métodos.

7.2.3. Srinivasa Ramanujan (1914)

Hemos visto un método, debido a Euler, para aproximar π . En este ejercicio consideramos otro, mucho más reciente y debido a S. Ramanujan.

Ejercicio 4.

1. Define una función de Sage `suma(N)` que devuelve el valor de la suma

$$\sum_{n=0}^{n=N} \frac{((2n)!)^3 \cdot (42n + 5)}{(n!)^6 \cdot (16^{(3n+1)})}.$$

2. Comprueba que $1/\text{suma}(N)$ se aproxima a π cuando N crece.
3. Intenta estimar, produciendo un programa adecuado, cuantas cifras correctas de π se obtienen cada vez que añadimos 10 sumandos más a `suma(N)`, es decir, cada vez que incrementamos N en 10 unidades.

7.2.4. Salamin y Brent (1976)

Este algoritmo difiere de los otros que estamos viendo en el hecho de que no se basa en una serie que converge a π sino en un proceso iterativo que produce una sucesión que converge a π . Entonces, es similar al algoritmo utilizado por Arquímedes.

Se parte de los valores iniciales $a_0 = 1$, $b_0 = \frac{\sqrt{2}}{2}$ y $s_0 = 1/2$ y se itera de acuerdo a las fórmulas

$$\begin{aligned} a_k &:= \frac{a_{k-1} + b_{k-1}}{2} \\ b_k &:= \sqrt{a_{k-1}b_{k-1}} \\ c_k &:= a_k^2 - b_k^2 \\ s_k &:= s_{k-1} - 2^k c_k \\ p_k &:= \frac{2a_k^2}{s_k} \end{aligned}$$

y p_k es una sucesión con límite π . Lo interesante de este algoritmo es que en cada paso de la iteración se DUPLICA el número de cifras correctas de π obtenidas, mientras que los algoritmos basados en series típicamente suman un cierto valor constante, que depende de la serie, al número de cifras correctas que teníamos cada vez que añadimos un sumando más.

Sin embargo, aunque este algoritmo es *en teoría* muy potente, cuando se implementa no es mejor que algoritmos eficientes basados en series debido simplemente a que el tiempo que tarda el ordenador en realizar cada paso de p_{k-1} a p_k crece mucho con k .

Ejercicio 5.

Implementa el algoritmo de Salamin y Brent y compara su eficiencia con la del método basado en la serie de Ramanujan.

7.2.5. Chudnovsky (199*)

Otra serie, una variante de la de Ramanujan debida a los hermanos Chudnovsky, que permite aproximar π de manera todavía más eficiente es

$$S(N) := \sum_{n=0}^{n=N} \frac{(-1)^n \cdot (6n)! \cdot (545140134n + 13591409)}{(3n)! \cdot (n!)^3 \cdot (640320^{3n})},$$

con la que se obtiene

$$\pi = \lim_{N \rightarrow \infty} \frac{426880 * \sqrt{10005}}{S(N)}.$$

Cuando programamos esta fórmula no conocemos *a priori* el número de sumandos que necesitamos para obtener una cantidad de dígitos de π correctos fijada de antemano. En consecuencia, nos conviene utilizar un bucle **while** y la única dificultad consiste en formular la condición de parada adecuada.

Definimos

```
def F(m):
    '''Lineas que terminan en \ continuan en la siguiente'''
    A = 545140134
    B = 13591409
    C = 640320
    return ((-1)^m*(factorial(6*m))*(A*m+B))/ \
            (factorial(3*m)*(factorial(m)^3)*(C^(3*m)))

def pi_chudnovski(ndigits):
    k = 0
    S = 0
    while 1:
        S += F(k)
        if floor(abs(10^ndigits*F(k))) == 0:
            break
        k += 1
    return (426880*sqrt(10005).n(digits=ndigits))/S,k
```

Estudia con cuidado este programa y, en particular,

1. Explica el uso de **while** 1: ... **if** ... :**break**.
2. ¿Cuál es la justificación de la condición de parada

$$\mathbf{floor}(\mathbf{abs}(10^{\text{ndigits}} * F(k))) == 0$$

El programa anterior no es muy eficiente, pero admite muchas mejoras. La primera consiste en observar que si escribimos

$$a_n := \frac{(-1)^n (6 * n)!}{(3n)!(n!)^3 640320^{3n}}, b_n := \frac{(-1)^n (6 * n)! n}{(3n)!(n!)^3 640320^{3n}},$$

tenemos

$$\pi = \frac{426880\sqrt{10005}}{B \sum_{n=0}^{\infty} a_n + A \sum_{n=0}^{\infty} b_n},$$

pero $b_n = n \cdot a_n$ y el cociente a_n/a_{n-1} vale

$$\frac{(-1)(6n)(6n-2)(6n-3)(6n-4)(6n-5)}{640320^3 n^3 (3n)(3n-1)(3n-2)}.$$

Entonces, si ya hemos calculado a_{n-1} , es fácil, es decir no requiere muchas multiplicaciones, obtener, a partir de a_{n-1} , los valores de b_{n-1} , a_n y b_n .

En resumen, podemos calcular cada término en la serie en función del anterior sin necesidad de repetir una inmensidad de multiplicaciones que ya se habían hecho.

Ejercicio 6.

Modifica el programa dado para incluir esta mejora.

Cuando se implementa este truco, y otros similares, y se programa en un lenguaje de programación compilado, como por ejemplo *C*, este método es enormemente eficiente y permite calcular millones de cifras de π en segundos. Como curiosidad puedes ejecutar las siguientes líneas en una terminal para ver un tal programa en acción:

```
1 gcc ~/Desktop/SAGE-noteb/bin/gmp-chudnovsky.c \
2     -lgmp -lm -O2 -o ~/Desktop/pi-chudnovsky
3 chmod +x ~/Desktop/pi-chudnovsky
4 ~/Desktop/pi-chudnovsky 10000000 1 > ~/Desktop/pi-diez-millones.txt
```

Ejercicio 7. *Modifica el programa obtenido en el ejercicio anterior usando **cython** y **numpy** y compara la eficiencia del programa resultante con la de **gmp-chudnovsky.c**.*

*¿Hasta qué punto **cython+numpy** se acerca a *C*?*

7.2.6. Rabinowitz y Wagon (1995)

Otra serie que permite calcular dígitos de π de forma bastante eficiente es

$$\pi = \sum_{n=0}^{\infty} \frac{(n!)^2 2^{n+1}}{(2n+1)!}.$$

Ejercicio 8.

Implementa este cálculo en Sage , teniendo en cuenta que una simplificación como la de la sección anterior es posible, y trata de entender cuántos sumandos son necesarios para obtener cada cifra adicional de π .

7.2.7. ¿Cuál es la cifra que ocupa el lugar 1000001 en π ? (BBP)

Es cuando poco sorprendente que exista un algoritmo eficiente que permite responder a esa pregunta sin que sea necesario calcular las cifras anteriores. El cálculo se basa en la serie

$$\pi = \sum_{k=0}^{k=\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{4}{8k+6} \right), \quad (7.2)$$

y en las siguientes observaciones:

1. La parte más costosa del cálculo es el de las potencias de 16. Ésto sugiere que puede ser útil tratar de realizar este cálculo módulo enteros adecuadamente elegidos y usando el truco que ya conocemos para calcular rápido las potencias.
2. Si queremos calcular la cifra n -ésima, contando a partir del punto decimal, debemos multiplicar la expresión decimal de π por 10^n y la cifra que nos interesa es la primera cifra decimal del número que obtenemos.

Teniendo en cuenta que todos los términos de la serie tienen un factor de la forma $1/16^k$, es claro que multiplicar por potencias de 10 no es muy conveniente y debemos usar el sistema de numeración en base 16 (hexadecimal). Por tanto, para obtener la n -ésima cifra hexadecimal de π debemos multiplicar la serie por 16^n y observar la primera cifra después del punto.

3. Cuando queremos la cifra n -ésima observamos que en

$$16^n \pi = \sum_{k=0}^{k=\infty} 16^{n-k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{4}{8k+6} \right), \quad (7.3)$$

hay dos clases de sumandos: sumandos en los que el exponente de 16 verifica $n - k \geq 0$ y el resto de los sumandos.

Consideremos un sumando como $16^{n-k}/(8k+1)$ con $n - k \geq 0$: como sólo nos interesa la parte decimal del resultado podemos sustituir esta fracción por $(16^{n-k} \% (8k+1))/(8k+1)$ y obtenemos la misma contribución al resultado final.

Lo importante aquí es que, como ya sabemos, el cálculo de potencias módulo un entero se puede hacer de forma muchísimo más rápida que en los números enteros.

El resto de los sumandos contribuyen cantidades muy pequeñas a la suma, y enseguida podemos dejar de sumarlos porque no afectan al resultado final.

```
def F0(j,m):
    '''Sumandos con m-k no negativo'''
    S = RR(0.0)
    k = 0
    while k <= m:
        r = 8*k+j
        S += RR(power_mod(16,n-k,r)/r) - \
            floor(RR(power_mod(16,n-k,r)/r))
        k += 1
    return RR(S)
```

```
def F1(j,m):
    '''Resto de los sumandos'''
    S = RR(0.0)
    k = m+1
    while 1:
```

```

    r = 8*k+j
    nS = S+ RR(16^(m-k)/r)
    if S == nS:
        break
    else:
        S = nS
    k += 1
return RR(S)

def S(j,m):
    return RR(F0(j,m)+F1(j,m))
def cifra_pi(n):
    n -= 1
    x = (4*RR(S(1,n))-2*RR(S(4,n))-RR(S(5,n))- \
        RR(S(6,n)))
    return (x-floor(x)).str(base=16)

```

COMENTARIOS:

1. Este algoritmo fué descubierto por Bayley, Borwein y Plouffe en 1997. Puedes ver el artículo original en [este enlace](#).
2. Estudia cuidadosamente este programa, fijándote en particular en
 - a) el uso que se hace de los números reales de precisión doble (**RR**).
 - b) la forma de transformar el resultado a hexadecimal en la última línea.
3. Antes del descubrimiento de esta clase de algoritmos se pensaba que para calcular la n -ésima cifra de π había que efectuar todo el cálculo, por ejemplo sumando una serie, con más de n dígitos de precisión y, por supuesto, durante el proceso se calculaban todas las cifras anteriores.

En cambio, en este algoritmo se usan números reales de precisión doble independientemente de lo grande que sea n . La cantidad de memoria RAM que usa el programa es muy reducida porque se reutiliza todo el tiempo la misma memoria.

4. Debe ser posible obtener un rendimiento mayor del algoritmo programándolo en C, o usando **cython**.

Ejercicio 9.

1. Hay otras constantes de las que podemos calcular la cifra n -ésima sin calcular las anteriores. Por ejemplo, se puede usar la serie

$$\log(2) = \sum_{k=1}^{\infty} \frac{1}{k2^k},$$

para calcular cifras del logaritmo neperiano de 2. Modifica el programa anterior para adaptarlo a este caso.

¿Cuál podría ser la forma general de series a las que se les puede aplicar este método?

2. Intenta adaptar este programa para calcular, de manera eficiente (i.e. evitando en lo posible repetir cálculos que ya se han hecho), N cifras consecutivas de π empezando en la que ocupa el lugar n_0 .

7.2.8. Aproximaciones racionales de π

Debemos saber que π no es un número racional, es decir, su expresión como un decimal no es periódica. Sin embargo, hay fracciones que aproximan bastante bien el valor de π .

Cuando calculamos cualquier valor aproximado de π , como un decimal con un cierto número de cifras después del punto, estamos calculando un número racional próximo a π , pero se trata de números racionales con denominador una potencia de 10. También podemos usar series de sumandos racionales que aproximan π lo que nos da, al efectuar la suma finita, números racionales próximos a π , y ahora el denominador no será, en general, una potencia de diez.

En este ejercicio buscamos la mejor aproximación racional de π con un número, acotado a priori, de cifras en el denominador. En estas condiciones, podemos intentar un enfoque de *fuerza bruta* ya que, fijado un entero positivo k , hay un número finito de fracciones a/b tales que, por ejemplo, $3.14 < a/b < 3.15$ y $0 < b < 10^k$ (¿por qué?).

Algunas de esas aproximaciones son muy buenas, en el sentido de que teniendo un denominador no muy grande nos dan un número inesperadamente grande de cifras correctas de π .

Ejercicio 10.

1. Define una función de Sage que encuentre, y devuelva, la fracción, con denominador de como máximo k cifras (en base 10), que mejor aproxime a π .

NO SE PERMITE usar el método `exact_rational()` de Sage, que más o menos puede hacer lo que se pide en el ejercicio. SÍ SE PUEDE usar el valor de π que tiene internamente Sage.

¿Cuántos bucles tendrá tu función?

2. Jugando con los rangos de los bucles, pero sin usar explícitamente los resultados mencionados más abajo, trata de conseguir un programa lo más eficiente posible.

ALGUNOS RESULTADOS:

- a) Con denominador de 3 cifras, como máximo, se obtiene la fracción $355/113$, con un error del orden de 3×10^{-7} .
- b) Con denominador de 4 cifras, como máximo, se obtiene la misma fracción $355/113$ como la mejor aproximación. En mi máquina tarda 10 segundos.
- c) Con denominador de 5 cifras, como máximo, se obtiene la fracción $312689/99532$, con un error del orden de 3×10^{-11} . En mi máquina tarda 100 segundos.

7.3. Fracciones continuas

7.3.1. Las fracciones continuas aparecen al dividir

Consideramos la fracción $\frac{55}{43}$. Al aplicar el algoritmo de Euclides para calcular el máximo común divisor de 55 y 43, obtenemos como pasos intermedios las expresiones

$$\begin{aligned} 55 &= 1 \cdot 43 + 12, \\ 43 &= 3 \cdot 12 + 7, \\ 12 &= 1 \cdot 7 + 5, \\ 7 &= 1 \cdot 5 + 2, \\ 5 &= 2 \cdot 2 + 1, \\ 2 &= 2 \cdot 1 + 0. \end{aligned}$$

Los números 1, 3, 1, 1, 2, 2 son los cocientes parciales del algoritmo. Utilizando esta información podemos escribir la fracción $\frac{55}{43}$ de una forma curiosa:

$$\begin{aligned} \frac{55}{43} &= 1 + \frac{12}{43} = 1 + \frac{1}{\frac{43}{12}} = 1 + \frac{1}{3 + \frac{7}{12}} = 1 + \frac{1}{3 + \frac{1}{1 + \frac{5}{7}}} = 1 + \frac{1}{3 + \frac{1}{1 + \frac{1}{2 + \frac{2}{5}}}} = \\ &= 1 + \frac{1}{3 + \frac{1}{1 + \frac{1}{1 + \frac{1}{2 + \frac{1}{2}}}}}. \end{aligned}$$

La expresión de más a la derecha de esta cadena de igualdades es lo que se conoce como una fracción continua (simple) finita. Para describirla de una forma más compacta, utilizaremos en lo que sigue cualquiera de las dos notaciones siguientes:

$$1 + \frac{1}{3 + \frac{1}{1 + \frac{1}{1 + \frac{1}{2 + \frac{1}{2}}}}} = 1 + \frac{1}{3 + \frac{1}{1 + \frac{1}{1 + \frac{1}{2 + \frac{1}{2}}}}} = [1; 3, 1, 1, 2, 2].$$

No hay nada de especial en los números 55 y 43. Podemos seguir el mismo procedimiento con dos enteros cualesquiera, a , b con $a \neq 0$ para escribir b/a como una fracción continua finita.

Observación. La representación de un racional como fracción continua finita no es única, ya que, si $a_n \geq 2$, tenemos que

$$[a_0; a_1, \dots, a_n] = [a_0; a_1, \dots, a_n - 1, 1],$$

mientras que si $a_n = 1$ podemos escribir

$$[a_0; a_1, \dots, a_{n-1}, 1] = [a_0; a_1, \dots, a_{n-1} + 1].$$

Sin embargo, estas son las únicas ambigüedades posibles y, si hacemos la hipótesis de que el último cociente parcial sea siempre mayor que 1, $a_n > 1$, entonces la representación de un número racional como fracción continua es única.

7.3.2. Las fracciones continuas aparecen al resolver ecuaciones

Consideramos la ecuación $x^2 - x - 1 = 0$. Su única solución positiva es la razón áurea, $\Phi = \frac{1+\sqrt{5}}{2}$. Podemos reescribir $\Phi^2 - \Phi - 1 = 0$ como $\Phi = 1 + \frac{1}{\Phi}$. Sustituyendo la Φ del denominador por $\Phi = 1 + \frac{1}{\Phi}$ obtenemos

$$\Phi = 1 + \frac{1}{1 + \frac{1}{\Phi}}.$$

Repetiendo este proceso de sustitución “hasta el infinito”, podemos escribir

$$\text{“}\Phi = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \ddots}}}}\text{.”}$$

El lado derecho de esta expresión es un ejemplo de una fracción continua infinita. ¿Por qué hemos puesto la igualdad entre comillas? Porque tenemos que dar sentido al lado derecho de la expresión, a la fracción continua infinita.

7.3.3. Fracciones continuas infinitas

Sea a_0 un entero arbitrario (positivo, negativo o cero), y $\{a_j\}_{j=1}^{\infty}$, una sucesión de enteros positivos. Llamamos n -ésimo convergente de la fracción continua infinita (simple) $[a_0; a_1, a_2, \dots]$ a la fracción continua finita $c_n = [a_0; a_1, \dots, a_n]$. Nótese que c_n está bien definido, y que es un número racional. Si el límite $\lim_{n \rightarrow \infty} c_n$ existe, decimos que la fracción continua infinita $[a_0; a_1, a_2, \dots]$ converge, y denotamos $[a_0; a_1, a_2, \dots] = \lim_{n \rightarrow \infty} c_n$.

Un sencillo argumento de inducción permite demostrar que los convergentes $c_n = [a_0; a_1, \dots, a_n]$ verifican

$$c_n = \frac{p_n}{q_n},$$

donde las sucesiones $\{p_n\}$ y $\{q_n\}$ vienen dadas por las relaciones de recurrencia de Wallis-Euler,

$$\begin{aligned} p_0 &= a_0, & p_1 &= a_0 a_1 + 1, & p_n &= a_n p_{n-1} + p_{n-2}, & n &\geq 2 \\ q_0 &= 1, & q_1 &= a_1, & q_n &= a_n q_{n-1} + q_{n-2}, & n &\geq 2. \end{aligned}$$

No es difícil demostrar a partir de aquí que

$$\begin{aligned} p_n q_{n-1} - q_n p_{n-1} &= (-1)^{n-1}, \\ p_n q_{n-2} - q_n p_{n-2} &= (-1)^n a_n, \end{aligned} \quad n = 1, 2, \dots$$

Como consecuencia inmediata se obtiene que p_n y q_n son coprimos. Además, aplicando sucesivas veces estas identidades se obtiene que

$$c_n - c_{n-1} = \frac{(-1)^{n-1}}{q_n q_{n-1}}, \quad n \geq 1, \quad c_n - c_{n-2} = \frac{(-1)^n a_n}{q_n q_{n-2}}, \quad n \geq 2.$$

Por otra parte, $q_n < q_{n+1}$ para todo $n \geq 0$, y además $\lim_{n \rightarrow \infty} q_n = \infty$, por lo que concluimos que las fracciones continuas infinitas (simples) siempre convergen a un cierto $\alpha = \lim_{n \rightarrow \infty} c_n$, y que los convergentes satisfacen

$$c_0 < c_2 < c_4 < \dots < c_{2n} < \dots < \alpha < \dots < c_{2n-1} < \dots < c_5 < c_3 < c_1.$$

Se tiene además la estimación

$$|\alpha - c_n| < \frac{1}{q_n q_{n+1}} < \frac{1}{q_n^2},$$

válida para todos los valores de n , lo que implica que α no puede ser racional. Así pues, una fracción continua (simple) es racional si y sólo si es finita.

Ejemplo. En particular, la fracción continua $\Phi := [1; 1, 1, 1, \dots]$ converge. ¿A qué? Nótese que

$$\Phi = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \dots}}}} = 1 + \frac{1}{\Phi} \quad \Rightarrow \quad \Phi = 1 + \frac{1}{\Phi}.$$

También podemos obtener esta fórmula a partir de los convergentes. En efecto, el n -ésimo convergente de Φ es

$$c_n = 1 + \frac{1}{1 + \frac{1}{1 + \frac{\ddots}{1 + \frac{1}{1 + 1}}}} = 1 + \frac{1}{c_{n-1}}.$$

Por lo tanto, si denotamos por Φ al $\lim_{n \rightarrow \infty} c_n$, que sabemos que existe, tomando $n \rightarrow \infty$ en ambos miembros de $c_n = 1 + \frac{1}{c_{n-1}}$, obtenemos que $\Phi = 1 + \frac{1}{\Phi}$, igual que antes. Concluimos que Φ es la única solución positiva de $x^2 - x - 1 = 0$, es decir,

$$\Phi = \frac{1 + \sqrt{5}}{2}.$$

7.3.4. Algoritmo canónico

¿Cómo construir la expansión en fracciones continuas de un número real? Ya sabemos cómo hacerlo para números racionales; el mismo método, interpretado adecuadamente, funcionará para irracionales, de manera que vamos a revisarlo.

Consideramos el número $\frac{157}{68} = [2; 3, 4, 5]$. Veamos cómo se obtiene su expansión en fracciones continuas. En primer lugar escribimos $\xi_0 := \frac{157}{68}$ como

$$\xi_0 = 2 + \frac{1}{\xi_1}, \quad \text{donde } \xi_1 = \frac{68}{21} > 1.$$

En particular,

$$a_0 = 2 = \lfloor \xi_0 \rfloor,$$

donde, para cada número real x , $\lfloor x \rfloor$ denota al mayor entero menor o igual que x . Seguidamente, escribimos $\frac{68}{21}$ como

$$\xi_1 = 3 + \frac{1}{\xi_2}, \quad \text{donde } \xi_2 = \frac{21}{5} > 1.$$

En particular,

$$a_1 = 3 = \lfloor \xi_1 \rfloor,$$

En tercer lugar, escribimos

$$\xi_2 = \frac{21}{5} = 4 + \frac{1}{\xi_3}, \quad \text{donde } \xi_3 = 5 > 1.$$

En particular,

$$a_2 = 4 = \lfloor \xi_2 \rfloor,$$

Finalmente, $a_3 = \lfloor \xi_3 \rfloor = \xi_3$ no se puede descomponer más, de manera que *paramos* aquí. Por consiguiente,

$$\frac{157}{68} = \xi_0 = 2 + \frac{1}{\xi_1} = 2 + \frac{1}{3 + \frac{1}{\xi_2}} = 2 + \frac{1}{3 + \frac{1}{4 + \frac{1}{\xi_3}}} = 2 + \frac{1}{3 + \frac{1}{4 + \frac{1}{5}}}$$

Acabamos de encontrar la fracción continua canónica (simple) de $157/68$. Nótese que acabamos con el número 5, que es mayor que 1; este será siempre el caso cuando apliquemos el procedimiento a un número racional no entero.

Podemos seguir exactamente el mismo procedimiento para los números irracionales:

Sea ξ un número irracional. Hacemos $\xi_0 = \xi$ y definimos $a_0 := \lfloor \xi_0 \rfloor \in \mathbb{Z}$. Entonces, $0 < \xi_0 - a_0 < 1$, de forma que podemos escribir

$$\xi_0 = a_0 + \frac{1}{\xi_1}, \quad \text{donde } \xi_1 := \frac{1}{\xi_0 - a_0} > 1.$$

Nótese que ξ_1 es irracional. En segundo lugar, definimos $a_1 := \lfloor \xi_1 \rfloor \in \mathbb{N}$. Entonces $0 < \xi_1 - a_1 < 1$, de forma que podemos escribir

$$\xi_1 = a_1 + \frac{1}{\xi_2}, \quad \text{donde } \xi_2 := \frac{1}{\xi_1 - a_1} > 1.$$

Nótese que ξ_2 es irracional. En tercer lugar, definimos $a_2 := \lfloor \xi_2 \rfloor \in \mathbb{N}$. Entonces $0 < \xi_2 - a_2 < 1$, de forma que podemos escribir

$$\xi_2 = a_2 + \frac{1}{\xi_3}, \quad \text{donde } \xi_3 := \frac{1}{\xi_2 - a_2} > 1.$$

Nótese que ξ_3 es irracional. Podemos continuar este procedimiento “hasta el infinito”, creando una sucesión de números reales $\{\xi_n\}_{n=0}^{\infty}$ de números reales con $\xi_n > 0$ para $n \geq 1$ llamada la sucesión de cocientes completos de ξ , y una sucesión $\{a_n\}_{n=0}^{\infty}$ de enteros con $a_n > 0$ para $n \geq 1$ llamada la sucesión de cocientes parciales de ξ , tales que

$$\xi = a_n + \frac{1}{\xi_{n+1}}, \quad n = 0, 1, 2, 3, \dots$$

Por consiguiente,

$$\xi = \xi_0 = a_0 + \frac{1}{\xi_1} = \xi_0 = a_0 + \frac{1}{a_1 + \frac{1}{\xi_2}} = \dots = "a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \frac{1}{a_4 + \ddots}}}}.$$

No hemos demostrado que ξ sea igual a la fracción continua infinita de la derecha (de ahí las comillas), pero esta igualdad es cierta.

Ejercicio 11.

1. Escribir un código que, tomando un $r > 1$ y un número dado k de pasos, halle los $[a_0, \dots, a_k]$ de la fracción continua de r .
2. Para el caso de un racional $r = n/m > 1$, hacer una variante del código anterior que tome los enteros n, m como datos y devuelva la lista completa $[a_0, \dots, a_k]$, hasta el fin del desarrollo.
3. Escribir otra función que tome los $[a_0, \dots, a_k]$ y produzca las fracciones (convergentes) $c_n = p_n/q_n$, $n = 0, \dots, k$.
4. Escribir una función que tome un $r > 1$ y una tolerancia `tol`, y devuelva una fracción que aproxime r con un error menor que `tol`.
5. Comprueba que las convergentes c_k tienen la siguiente propiedad: el error de la aproximación c_k es menor que el error de cualquier aproximación con un denominador menor que el denominador de q_k . Efectúa la comprobación con al menos un número algebraico y otro trascendente, para al menos las cuatro primeras aproximaciones c_k .

7.4. Fracciones continuas periódicas

Sabemos que, entre los números reales, solo los racionales tienen un desarrollo decimal periódico. Podemos preguntarnos qué hay sobre este asunto en los desarrollos en fracción continua: ¿qué reales tienen fracciones continuas periódicas?

Hemos visto ya algún ejemplo, la razón áurea con desarrollo

$$\Phi = \frac{1 + \sqrt{5}}{2} = [1; 1, 1, 1, 1, \dots]$$

que denotaremos como $[\bar{1}]$ para indicar que el 1 se repite.

Si tomamos la raíz cuadrada de 8 se llega a $\sqrt{8} = [2; 1, 4, 1, 4, \dots] = [2; \overline{1, 4}]$, donde tenemos que se repite indefinidamente el bloque 1, 4. En efecto, $\lfloor \sqrt{8} \rfloor = 2$ de manera que:

$$\begin{aligned} \sqrt{8} &= \mathbf{2} + (\sqrt{8} - 2) \text{ con } \sqrt{8} - 2 < 1 \\ \frac{1}{\sqrt{8} - 2} &= \frac{\sqrt{8} + 2}{8 - 4} = \frac{\sqrt{8} + 2}{4} \text{ donde } 1 < \frac{\sqrt{8} + 2}{4} < 2 \text{ y así} \\ \frac{\sqrt{8} + 2}{4} &= 1 + \left(\frac{\sqrt{8} + 2}{4} - 1 \right) = \mathbf{1} + \frac{\sqrt{8} - 2}{4} \text{ con } \frac{\sqrt{8} - 2}{4} < 1 \\ \frac{1}{\frac{\sqrt{8} - 2}{4}} &= \frac{4(\sqrt{8} + 2)}{8 - 4} = \sqrt{8} + 2 = \mathbf{4} + (\sqrt{8} - 2) \text{ con } \sqrt{8} - 2 < 1. \end{aligned}$$

Y hemos llegado a un valor ya antes visitado en el algoritmo: $\frac{1}{\sqrt{8} - 2} = \dots$

Antes de ver cómo recuperar $\sqrt{8}$ a partir de esta fracción continua, veamos otro ejemplo, esta vez con periodo puro.

Desarrollemos nuestro algoritmo para $\sqrt{7} + 2$:

$$\begin{aligned}\sqrt{7} + 2 &= \textcolor{red}{4} + (\sqrt{7} - 2) = \textcolor{red}{4} + \frac{1}{\frac{1}{\sqrt{7}-2}} \\ \frac{1}{\sqrt{7}-2} &= \frac{\sqrt{7}+2}{3} = \textcolor{red}{1} + \frac{\sqrt{7}-1}{3} = \textcolor{red}{1} + \frac{\frac{3}{\sqrt{7}-1}}{\frac{3}{\sqrt{7}-1}} \\ \frac{3}{\sqrt{7}-1} &= \frac{3(\sqrt{7}+1)}{6} = \frac{\sqrt{7}+1}{2} = \textcolor{red}{1} + \frac{\sqrt{7}-1}{2} = \textcolor{red}{1} + \frac{\frac{2}{\sqrt{7}-1}}{\frac{2}{\sqrt{7}-1}} \\ \frac{2}{\sqrt{7}-1} &= \frac{2(\sqrt{7}+1)}{6} = \frac{\sqrt{7}+1}{3} = \textcolor{red}{1} + \frac{\sqrt{7}-2}{3} = \textcolor{red}{1} + \frac{\frac{3}{\sqrt{7}-2}}{\frac{3}{\sqrt{7}-2}} \\ \frac{3}{\sqrt{7}-2} &= \frac{3(\sqrt{7}+2)}{3} = \sqrt{7} + 2 \text{ (igual que empezamos)}\end{aligned}$$

y así: $\sqrt{7} + 2 = [4, 1, 1, 1]$.

Podemos recuperar este último valor a partir de la expresión $[4, 1, 1, 1]$. Basta resolver la siguiente ecuación:

$$x = [4, 1, 1, 1, x]. \quad (7.4)$$

Desarrollando el miembro derecho se obtiene

$$\begin{aligned}[4, 1, 1, 1, x] &= \textcolor{red}{4} + \frac{1}{\textcolor{red}{1} + \frac{1}{\textcolor{red}{1} + \frac{1}{\textcolor{red}{1} + \frac{1}{x}}}} = \textcolor{red}{4} + \frac{1}{\textcolor{red}{1} + \frac{1}{\textcolor{red}{1} + \frac{x}{x+1}}} = \textcolor{red}{4} + \frac{1}{\textcolor{red}{1} + \frac{x+1}{2x+1}} = \\ &= \textcolor{red}{4} + \frac{2x+1}{3x+2} = \frac{14x+9}{3x+2}.\end{aligned}$$

La ecuación (7.4) es por tanto una ecuación cuadrática:

$$x = \frac{14x+9}{3x+2} \iff 3x^2 - 12x - 9 = 0$$

y su raíz positiva, $\frac{12+\sqrt{144+108}}{6} = 2 + \sqrt{7}$, es el número buscado.

Con una pequeña variante también se puede recuperar el irracional correspondiente a un desarrollo periódico no puro, como $[2; \overline{1, 4}]$. Basta observar que, si $x = [\overline{1, 4}]$, entonces $[2; \overline{1, 4}] = [2; x] = 2 + \frac{1}{x}$. Si desarrollamos $x = [1; 4, x]$ llegamos a $x = \frac{5x+1}{4x+1}$, de donde a la ecuación cuadrática $4x^2 - 4x - 1 = 0$. La raíz positiva es $\frac{\sqrt{2}+1}{2}$, y

$$2 + \frac{2}{1 + \sqrt{2}} = \frac{4 + 2\sqrt{2}}{1 + \sqrt{2}} = (2\sqrt{2} + 4)(\sqrt{2} - 1) = 2\sqrt{2} = \sqrt{8}.$$

Acabamos de ver cómo una ecuación del tipo:

$$x = [\overline{a_0, a_1, \dots, a_m}] = [a_0; a_1, \dots, a_m, x]$$

nos ha llevado a una ecuación cuadrática, de coeficientes enteros si los a_i son enteros. Si los a_i son positivos, la solución positiva de esta ecuación cuadrática es el irracional con desarrollo en fracción continua periódico $[\overline{a_0, a_1, \dots, a_m}]$.

El caso periódico más general no es muy distinto, y podemos calcular el irracional con desarrollo

$$[a_0; a_1, \dots, a_\ell, \overline{b_0, b_1, \dots, b_m}]$$

a partir del irracional raíz de la ecuación cuadrática

$$x = [b_0, b_1, \dots, b_m, x].$$

Aunque son solo un par de ejemplos, este es un resultado general demostrado por Lagrange (1736–1813). Antes de enunciarlo, algo de notación. Definimos:

$$\mathbb{Z}[\sqrt{d}] := \{a + b\sqrt{d} : a, b \in \mathbb{Z}\}$$

$$\mathbb{Q}[\sqrt{d}] := \{a + b\sqrt{d} : a, b \in \mathbb{Q}\}$$

Dado $\xi = a + b\sqrt{d}$ en $\mathbb{Z}[\sqrt{d}]$ o $\mathbb{Q}[\sqrt{d}]$, se define su conjugado como $\bar{\xi} := a - b\sqrt{d}$. Con las operaciones obvias, $\mathbb{Z}[\sqrt{d}]$ es un anillo conmutativo, y $\mathbb{Q}[\sqrt{d}]$ un cuerpo. Además, la conjugación preserva las propiedades algebraicas: si $\alpha, \beta \in \mathbb{Q}[\sqrt{d}]$, entonces

$$\overline{\alpha \pm \beta} = \bar{\alpha} \pm \bar{\beta}, \quad \overline{\alpha \cdot \beta} = \bar{\alpha} \cdot \bar{\beta}.$$

Obsérvese que si d es un entero positivo que no es un cuadrado ($\sqrt{d} \notin \mathbb{Z}$), y $b \neq 0$, $a + b\sqrt{d}$ es un irracional. Estos irracionales se dicen cuadráticos pues se pueden poner como solución de una ecuación cuadrática con coeficientes enteros:

$$\xi = r + s\sqrt{d} \in \mathbb{Q}[\sqrt{d}] \text{ es solución de: } x^2 - 2rx + (r^2 - s^2d) = 0$$

Si multiplicamos por el mínimo común múltiplo de los denominadores de $2r$ y $r^2 - s^2d$, se tiene un polinomio cuadrático con coeficientes enteros.

Ahora ya podemos enunciar el resultado de Lagrange:

Teorema 1 (Lagrange). *Una fracción continua simple infinita es un irracional cuadrático si y solo si es periódica.*

Entre los irracionales cuadráticos los hay con fracción continua periódica pura, ¿quiénes son?

Teorema 2 (Galois). *Un irracional cuadrático ξ es puramente periódico si y solo si*

$$\xi > 1 \quad \text{y} \quad -1 < \bar{\xi} < 0.$$

Un caso especialmente llamativo es el de las raíces cuadradas de los enteros (no cuadrados).

Teorema 3. *Sea d un entero positivo que no es un cuadrado. La fracción continua para \sqrt{d} es de la forma*

$$[a_0; \overline{a_1, \dots, a_{n-1}, a_n}]$$

con $a_n = 2a_0$ y $a_k \leq a_0$ para $1 \leq k < n$.

Además, si $m \geq 1$ y $x = p_{mn-1}$ e $y = q_{mn-1}$ son el numerador y el denominador de la $(mn-1)$ -ésima convergente de \sqrt{d} , entonces:

$$x^2 - dy^2 = \pm 1.$$

OBSERVACIÓN: Es un cálculo directo el comprobar que si x_1, y_1 son tales que $x_1^2 - dy_1^2 = -1$, al tomar x, y tales que $x + y\sqrt{d} = (x_1 + y_1\sqrt{d})^2$, se tiene que $x^2 + dy^2 = 1$.

Una aplicación directa de este resultado la encontramos en la resolución de las llamadas ecuaciones de Pell.

7.4.1. Ecuaciones de Pell

Una ecuación de Pell es una ecuación diofántica de la forma:

$$x^2 - dy^2 = 1 \quad (7.5)$$

donde d es un entero, y para la que se buscan soluciones enteras positivas. Siempre se tiene la solución trivial $x = 1, y = 0$. Si d es un cuadrado, no existen soluciones enteras no triviales a la ecuación de Pell. Si d no es un cuadrado, existen infinitas soluciones.

Obsérvese que, si $y \neq 0$, podemos reescribir la ecuación como

$$\frac{x^2}{y^2} - d = \frac{1}{y^2}$$

y así, $\frac{x}{y}$ se acerca a \sqrt{d} . Del Teorema 3 se deduce el siguiente algoritmo para encontrar una solución a la ecuación de Pell (7.5).

Resolviendo $x^2 - dy^2 = 1$

1. Calcular $a_0 = \lfloor \sqrt{d} \rfloor$.
2. Calcular la fracción continua $[a_0; a_1, a_2, \dots]$ para \sqrt{d} . Parar cuando $a_n = 2a_0$.
3. Sea $x_1 = p_{n-1}$ e $y_1 = q_{n-1}$ (el numerador y el denominador de la convergente $n - 1$).
4. Si $x_1^2 - dy_1^2 = 1$, tomar $x = x_1$ e $y = y_1$ y parar.
5. Si $x_1^2 - dy_1^2 = -1$, calcular $x + y\sqrt{d} = (x_1 + y_1\sqrt{d})^2$. Entonces $x^2 - dy^2 = 1$, y parar.

Ejemplo 1. Se quiere encontrar una solución entera no trivial a la ecuación

$$x^2 - 13y^2 = 1.$$

El inicio del desarrollo en fracción continua de $\sqrt{13}$ es:

$$[3; 1, 1, 1, 1, 6, \dots].$$

Puesto que 6 es el doble de 3, calculamos la convergente:

$$[3; 1, 1, 1, 1] = \frac{18}{5}$$

y ocurre que $18^2 - 13 \cdot 5^2 = -1$. Calculamos ahora:

$$(18 + 5\sqrt{13})^2 = 649 + 180\sqrt{13}.$$

y vemos que $649^2 - 13 \cdot 180^2 = 1$, con lo que hemos encontrado una solución:

$$x = 649, y = 180.$$

Ejercicio 12.

1. Resolver, con ayuda de sage, la ecuación $x = [4, 1, 2, 1, x]$. SUGERENCIA: estudiar y utilizar la función `solve()`.
2. Encontrar, con ayuda de sage, el irracional cuadrático con fracción continua $[7; \overline{2, 1, 3, 1, 2, 8}]$.
3. Encontrar las convergentes $p_0/q_0, p_1/q_1, p_2/q_2$ y p_3/q_3 del desarrollo en fracción continua de 1.234567 y comprobar que

$$\left| 1.234567 - \frac{p_n}{q_n} \right| < \frac{1}{q_n q_{n+1}}$$

para $n = 0, 1, 2, 3$.

4. Encontrar a, b menores que 100 tales que

$$\left| .123456789 - \frac{a}{b} \right| < 10^{-8}.$$

5. Encontrar a, b con $0 < b < 20$ tales que

$$\left| \frac{3141}{5926} - \frac{a}{b} \right| < 0.001.$$

6. a) Encontrar la fracción continua de $\sqrt{7}$ y una solución no trivial de $x^2 - 7y^2 = 1$.

b) Encontrar una solución no trivial de $x^2 - 109y^2 = 1$.

Ejercicio 13. (Problema 94 del proyecto Euler)

Los apartados 1 y 2 contienen información útil para resolver el tercero, pero no forman parte del ejercicio.

1. Solución fundamental a la ecuación de Pell $x^2 - dy^2 = 1$ por medio de fracciones continuas

Sea $\{p_i/q_i\}$ la sucesión de convergentes de la fracción continua para \sqrt{d} . Entonces el par no trivial (x_1, y_1) que resuelve la ecuación de Pell minimizando x satisface $x_1 = p_i$, $y_1 = q_i$ para algún i . Este par se conoce como la solución fundamental. Así pues, la solución fundamental se puede encontrar calculando el desarrollo en fracción continua de \sqrt{d} y comprobando cada uno de los sucesivos convergentes hasta encontrar una solución de la ecuación de Pell.

2. Obtención de otras soluciones a partir de la solución fundamental

(Algoritmo de Brahmagupta) Una vez que se tiene la solución fundamental, todas las demás se pueden calcular mediante

$$x_k + y_k\sqrt{d} = (x_1 + y_1\sqrt{d})^k.$$

Equivalentemente, podemos obtener las demás soluciones por medio de las relaciones de recurrencia

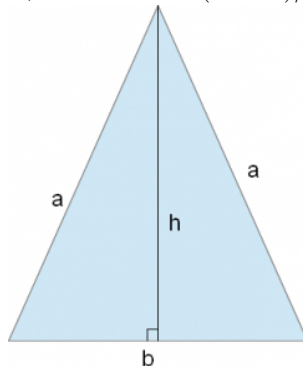
$$x_{k+1} = x_1x_k + dy_1y_k, \quad y_{k+1} = x_1y_k + y_1x_k.$$

3. Es fácil probar que no hay ningún triángulo equilátero con lados y área enteros. Sin embargo, el triángulo casi equilátero 5-5-6 tiene área 12.

Diremos que un triángulo es casi equilátero si tiene dos lados iguales y el tercero difiere de los anteriores en no más de una unidad.

Encontrar la suma de los perímetros de todos los triángulos casi equiláteros con lados y área enteros y cuyo perímetro no exceda a mil millones.

SUGERENCIA: Si a , b y h son como en la figura, y $b = a + 1$, entonces $x = (3a - 1)/2$ e $y = h$ satisfacen la ecuación de Pell $x^2 - 3y^2 = 1$. Análogamente, si $b = a - 1$, entonces $x = (3a + 1)/2$ e $y = h$ satisfacen la misma ecuación



7.5. Logaritmos

En Sage podemos calcular logaritmos mediante `log(x,base=b).n()`, y se verifica que, $b^{\wedge}(\text{log}(x,\text{base}=b).\text{n}())$ es aproximadamente x , por definición de logaritmo. Si no se especifica una base los logaritmos son naturales, neperianos, con base el número e .

1. Los logaritmos pueden servir para calcular el número de cifras de la expresión en base b de un entero. Si $b^k \leq N < b^{k+1}$ vemos, tomando logaritmos en base b , que $k \leq \log_b(N) < k + 1$, de forma que $k = \text{floor}(\text{log}(N, \text{base}=b).\text{n}())$. Entonces, la mayor potencia de b que aparece en la expresión en base b de N es b^k y el número de cifras es $k + 1$. Por ejemplo, podemos calcular el número de bits necesarios para representar un entero N calculando su logaritmo en base 2.
2. También podemos calcular la cifra dominante, manteniendo la notación anterior, el coeficiente de b^k :

Tenemos ahora $C \cdot b^k \leq N < (C + 1)b^k$, y tomando logaritmos en base b otra vez

$$\log_b(C) + k \leq \log_b(N) < \log_b(C + 1) + k,$$

de forma que, restando k a los tres términos y exponenciando en base b , obtenemos

$$C \leq b^{\log_b(N)-k} < C + 1.$$

Teniendo en cuenta que C es un dígito en base b , vemos que la parte entera por defecto de $b^{\log_b(N)-k}$ determina C .

3. La utilidad de este resultado radica en que hace posible calcular la cifra dominante de un entero N de la forma a^n , de manera muy eficiente y sin calcular N .
4. Ya usamos, en el ejercicio 5b del capítulo 6, logaritmos para calcular el n que corresponde a un número de Fibonacci F_n dado.

7.5.1. Un problema de Gelfand

Ejercicio 14.

Para cada $n = 1, 2, 3, 4, \dots$ consideramos la lista formada por los dígitos dominantes de b^n con b un dígito decimal excluyendo 0 y 1. Para cada n obtenemos una lista L_n con entradas dígitos decimales excluyendo el 0:

1. ¿Aparece el 9 como cifra dominante de un 2^n ?
2. La lista que obtenemos para $n = 1, [2, 3, 4, 5, 6, 7, 8, 9]$ ¿vuelve a aparecer para algún $n > 1$?
3. ¿Para algún n estará la lista L_n formada por 8 repeticiones del mismo dígito?
4. ¿Para algún n estará la lista L_n formada por los dígitos de un número primo de ocho cifras?
5. Estas preguntas, modificadas ligeramente, pueden enunciarse usando dígitos en otra base b , y, *a priori*, debería ser más fácil que tengan respuesta afirmativa con bases menores que 10. Estudia las modificaciones necesarias en los enunciados y trata de determinar si la respuesta es afirmativa para estos enunciados modificados.

Este ejercicio se puede hacer porque disponemos de una manera rápida, usando logaritmos, para calcular la cifra dominante de un entero b^n , pero puede ocurrir que los n que buscamos con nuestro programa sean tan grandes que no podamos llegar a verlos, o simplemente que perdamos la paciencia y paremos el programa.

Para una solución ver la hoja [79-APROX-gelfand.ipynb](#).

7.6. Ceros de funciones

Newton encontró un método, todavía importante, para obtener valores aproximados de las soluciones de ecuaciones de la forma $F(x) = a$ sin más que suponer la función F suficientemente derivable. Comenzamos estudiando el caso particular en que la función es exponencial.

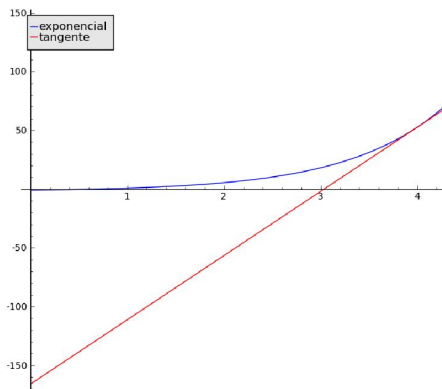
7.6.1. Existencia de logaritmos

Dado un número real $a > 0$, queremos resolver la ecuación $e^x = a$, es decir, queremos calcular, de manera aproximada, el logaritmo de a . Definimos la función $F_a(x) := e^x - a$, de forma que debemos ver que, para cada $a > 0$ hay un número x tal que $F_a(x) = 0$.

La idea es bastante simple: supongamos que $F_a(x_0) \neq 0$ y consideremos la recta tangente a la gráfica de la función F_a en el punto $(x_0, F_a(x_0)) = (x_0, e^{x_0} - a)$ que tiene ecuación

$$y - (e^{x_0} - a) = (F_a)'(x_0)(x - x_0) = e^{x_0}(x - x_0)$$

ya que la derivada de $F_a(x)$ en x_0 es e^{x_0} (“la derivada de la exponencial es ella misma”). La idea de Newton es que si cortamos la recta tangente con el eje de las X , con ecuación $y = 0$, el punto de corte puede estar más cerca de una verdadera solución de lo que estaba x_0 .



En la gráfica hemos tomado $a = 2$, es decir, buscamos un x tal que e^x valga 2, y empezamos con $x_0 = 4$ que está bastante lejos de ser una solución. Sin embargo, la recta tangente en $(4, e^4 - 2)$ corta al eje de las X en un punto $(x_1, 0)$ próximo a 3, que está más cerca de la verdadera solución que vemos en la gráfica que será un valor bastante cercano a cero.

Para acercarnos más a la solución bastará repetir el mismo procedimiento cambiando x_0 por la nueva solución aproximada x_1 . ¿Cuánto vale x_1 ?

Resolviendo el sistema formado por la ecuación de la recta tangente y la ecuación $y = 0$ obtenemos

$$x_1 = x_0 + \frac{e^{x_0} - a}{e^{x_0}}.$$

Definamos $T(x) := x + \frac{e^x - a}{e^x}$, de forma que $x_1 = T(x_0)$ y la sucesión de puntos que debería aproximarse a la solución es

$$x_0, T(x_0), T(T(x_0)), T(T(T(x_0))), \dots$$

Método de Newton

```
def Nw(x,a):
    return (x-((e^x-a)/e^x)).n()
```

```
def itera_hasta(ini,a,E):
    x = ini
    cont = 0
    while (abs((e^x-a).n())>E):
        x = Nw(x,a)
        cont +=1
    return x, log(a).n(), cont
```

```
itera_hasta(4,2,.000001)
```

```
(0.693147188845376, 0.693147180559945, 7)
```

Vemos que empezando en $x_0 = 4$ llegamos a calcular el logaritmo de 2 con un error menor que una millonésima ($E = 0.000001$) aplicando la función `Nw` siete veces, y tomando un valor de E todavía más pequeño podríamos conseguir una aproximación tan buena como queramos.

Como esto podemos hacerlo para cualquier $a > 0$, tenemos cierta evidencia acerca de la existencia de un logaritmo $\log(a)$ para cada $a > 0$.

7.6.2. Método de Newton

Queremos encontrar un número real x tal que $F(x) = 0$ y comenzamos con una solución aproximada x_0 . La ecuación de la recta tangente en el punto $(x_0, F(x_0))$ es

$$y - F(x_0) = F'(x_0)(x - x_0),$$

que corta al eje OX en $x_0 - \frac{F(x_0)}{F'(x_0)}$. Entonces, la transformación T que debemos iterar se define como

$$T(x) := x - \frac{F(x)}{F'(x)},$$

y esperamos que $T^n(x_0)$ se aproxime, cuando n tiende a infinito, a un punto en el que F se anula. ¿Bajo qué condiciones podemos asegurar que $T(x_0)$ es una mejor aproximación que x_0 ?

Por ejemplo, tenemos el siguiente resultado²

Sea f una función continua y derivable en el intervalo real I , $x_0 \in I$ un punto, y supongamos que existen constantes $C \geq 0$, $\lambda > 0$ tales que

1. $|f(x_0)| \leq (C/2)\lambda$, es decir, f toma valores pequeños cerca de x_0 .

2. Para todo par de puntos $x, y \in [x_0 - C, x_0 + C] \subset I$ se verifica

a) $|f'(x)| \geq 1/\lambda$, la derivada de f no se hace muy próxima a cero en el intervalo.

b) $|f'(x) - f'(y)| \leq (1/2)\lambda$, la derivada de f no varía mucho en el intervalo.

Entonces, existe una única solución, ξ_0 , de la ecuación $f(x) = 0$ en el intervalo $[x_0 - C, x_0 + C]$ y es el límite de la sucesión recursiva

$$x_{n+1} := x_n - \frac{f(x_n)}{f'(x_n)}.$$

²Puede verse la demostración en las páginas 60-61 de J.Dieudonne, *Cálculo Infinitesimal*.

7.6.3. Bisección

El método de bisección es una consecuencia del teorema de Bolzano: *si una función continua en un intervalo $[a, b]$ verifica $f(a) \cdot f(b) < 0$ entonces existe un $c \in [a, b]$ tal que $f(c) = 0$.*

Para utilizarlo observamos que si el intervalo $[a, b]$ verifica la condición $f(a) \cdot f(b) < 0$, entonces, salvo que $f(\frac{a+b}{2}) = 0$, al menos uno de los dos subintervalos, $[a, \frac{a+b}{2}]$ o $[\frac{a+b}{2}, b]$, también la verifica. Entonces, podemos aproximarnos a c subdividiendo el intervalo hasta que su longitud sea menor que un valor prefijado.

Ejercicio 15.

Implementa el método de bisección definiendo una función $\text{subint}(f, a, b)$ que devuelva uno de los subintervalos de $[a, b]$ que verifican la condición de Bolzano, y una segunda función $\text{iterador}(f, a, b, E)$ que subdivida el intervalo hasta que la diferencia en valor absoluto entre los extremos sea menor que E , y devuelva el último intervalo que ha calculado.

Este procedimiento devuelve un intervalo de longitud menor que un E prefijado y que, por el teorema de Bolzano, contiene un c en el que $f(c)$ es cero. Es de naturaleza bastante diferente al método de Newton ya que únicamente utiliza la continuidad de la función, un concepto muy posterior al de derivada.

7.7. Aproximación de funciones

Los polinomios, que se construyen simplemente usando las cuatro operaciones aritméticas, son sin duda las funciones más sencillas que se despachan. Resulta entonces natural tratar de usar polinomios para estudiar funciones más complejas, tratando de encontrar polinomios cuyos valores estén próximos a los de la función a estudiar.

Esto lleva, en primer lugar, a la idea de los *polinomios interpoladores*, *i.e. polinomios cuyos valores en un cierto número de puntos prefijados coinciden con los valores que toma la función que estamos estudiando*. Los polinomios interpoladores, que están forzados a coincidir con la función en un cierto número de puntos, tienen tendencia a alejarse bastante de la función en otros puntos, y, por tanto, no podemos decir que sean una *buena aproximación* de la función, pero tienen algunas aplicaciones interesantes.

Los polinomios interpoladores ya aparecieron como **ejemplo de problema de álgebra lineal**.

7.7.1. Interpolación de Lagrange

Un polinomio de grado n tiene $n + 1$ coeficientes y podemos fijar sus valores en $n + 1$ puntos distintos x_i . Mediante el “método de los coeficientes indeterminados”, suponiendo que el polinomio buscado es

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n,$$

y sustituyendo cada uno de los puntos x_i e igualando al valor prefijado en ese punto y_i , obtenemos un sistema lineal de ecuaciones, $n + 1$ ecuaciones con $n + 1$ incógnitas, cuya matriz es una [matriz de Vandermonde](#) con determinante siempre no nulo si los x_i son distintos entre sí.

Entonces, el polinomio interpolador existe y es la única solución del sistema lineal obtenido. En la práctica no resolvemos el sistema lineal ya que existe una fórmula, debida a Lagrange, que nos da directamente el [polinomio interpolador](#).

El polinomio interpolador se utiliza para estudiar, por el método de Kronecker, la [irreducibilidad de polinomios](#) con coeficientes enteros. También lo hemos usado para obtener [fórmulas para las sumas de potencias](#)

$$p(n, k) := 1^k + 2^k + 3^k + \cdots + n^k,$$

ya que una vez que sabemos, debido a que la suma es muy parecida a la integral definida

$$\int_1^n x^k dx,$$

que el resultado debe ser un polinomio de grado $k + 1$, es fácil obtener sus coeficientes por interpolación en $i = 1, 2, 3, \dots, k + 1$ ya que es fácil calcular directamente los valores $p(i, k)$.

7.7.2. Interpolación de Newton y polinomio de Taylor

Dados $n + 1$ valores distintos x_0, \dots, x_n distintos, y una función $f(x)$, se definen, recursivamente, las DIFERENCIAS DIVIDIDAS:

$$\begin{aligned} f[x_0, x_1] &= \frac{f(x_1) - f(x_0)}{x_1 - x_0} && \text{si } n = 1 \\ f[x_0, x_1, \dots, x_n] &= \frac{f[x_1, \dots, x_n] - f[x_0, x_1, \dots, x_{n-1}]}{x_n - x_0} && \text{para } n \geq 2. \end{aligned}$$

Se tiene el siguiente:

Teorema 1. Sean $(x_0, f(x_0)), (x_1, f(x_1)), \dots, (x_n, f(x_n))$, $n+1$ puntos distintos de la gráfica de f . El polinomio interpolador de grado n que pasa por estos $n+1$ puntos es

$$\begin{aligned} P_n(x) &= f(x_0) + \sum_{k=1}^n f[x_0, \dots, x_k] \prod_{i=0}^{k-1} (x - x_i) \\ &= f(x_0) + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \dots \\ &\quad \dots + f[x_0, x_1, \dots, x_n](x - x_0)(x - x_1) \cdots (x - x_{n-1}). \end{aligned}$$

El polinomio interpolador de Newton es el mismo que el de Lagrange, a fin de cuentas el polinomio interpolador es único, pero expresado en la base del espacio de polinomios de grado $\leq n$ dada por los polinomios $\prod_{i=0}^{k-1} (x - x_i)$ mientras que el de Lagrange está expresado en la base de monomios x^k .

Ejercicio 16. Encontrar, utilizando la forma de Newton, el polinomio de menor grado cuya gráfica pasa por los puntos

$$(-2, 26), (-1, 4), (1, 8), (2, -2).$$

Comparar el método utilizado con el anterior, de coeficientes indeterminados.

El aspecto quizá más interesante del polinomio interpolador de Newton es que se trata de una versión discreta del **polinomio de Taylor**: cuando todos los puntos x_i se hacen coincidir en un único punto z el polinomio interpolador nos da el polinomio de Taylor.

Sabemos que la derivada en x_0 permite aproximar una función $f(x)$, derivable en x_0 , mediante una función lineal

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \dots \quad (7.6)$$

De la misma forma, el polinomio de Taylor permite aproximar una función, derivable k veces, en la forma

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \dots + \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k + \dots$$

Estas aproximaciones de una función suficientemente derivable mediante un polinomio son *locales*: únicamente son válidas para x muy próximo a x_0 , pero en general no valen lejos de x_0 . Por ejemplo, la gráfica de la función lineal en (7.6) es la recta tangente, en x_0 , a la gráfica de la función f , y lejos de x_0 puede estar muy separada de la gráfica de f .

7.7.3. `find_fit`

Mediante la técnica de interpolación conseguimos un polinomio de grado mínimo tal que su gráfica pasa exactamente por un conjunto dado de puntos del plano. Esta condición es demasiado restrictiva y fuerza en ocasiones al polinomio interpolador a tener un comportamiento extraño.

Podemos relajarla buscando una función más sencilla, por ejemplo una recta, que pase *lo más cerca que sea posible* de todos los puntos, pero sin pasar necesariamente por ninguno de ellos. Se llama a esta técnica *regresión*, y consiste, en resumen, en lo siguiente:

1. Representamos gráficamente los puntos en el plano, `points(L)`, con L la lista de 2-tuplas formada por las coordenadas, y tratamos de *ver la forma de la curva que mejor se adapta*. Esto nos permite proponer un *modelo* $f(x, a_1, a_2, \dots)$ con x la variable independiente y los a_i parámetros del modelo.
2. Si los puntos son (x_i, y_i) definimos los residuos

$$r_i(a_1, a_2, \dots) := y_i - f(x_i, a_1, a_2, \dots),$$

que son función de los parámetros a_i .

3. Definimos el *error cuadrático medio* como

$$S(a_1, a_2, \dots) := \sum r_i(a_1, a_2, \dots)^2.$$

Al elevar al cuadrado los residuos eliminamos su signo, de forma que todos contribuyen incrementando $S(a_1, a_2, \dots)$.

4. Minimizamos la función $S((a_1, a_2, \dots))$, es decir, calculamos los valores de los parámetros que hacen mínimo el error cuadrático medio. Los valores que minimizan son los que declaramos *valores óptimos* para este problema.

Esta idea tiene muchas variantes, y puedes leer sobre ella en esta página de la [Wikipedia](#).

En Sage se puede ajustar un modelo a una lista de puntos mediante, por ejemplo,

```
var('a b')
modelo(x)=a*x+b
find_fit(L,modelo)
```

que nos devuelve los valores de los parámetros a, b que hacen mínimo el error cuadrático medio. En este caso el modelo es lineal, pero podemos definirlo mediante una función arbitraria.

En ocasiones el valor de alguno de los parámetros es muy pequeño, y debemos sospechar que el modelo correcto no depende de ese parámetro. Cambiamos la definición del modelo y comparamos, al menos mediante gráficas, los dos ajustes.

Esta instrucción es muy útil, y veremos ejemplos a lo largo del curso, para tratar de averiguar la forma de una función desconocida cuando hemos calculado en el ordenador sus valores para un cierto número de valores de su variable independiente. Un primer ejemplo aparece al resolver el tercer ejercicio en la [lista al final de este capítulo](#).

7.7.4. Otros find...

Aparte de `find_fit`, Sage dispone de, al menos otras 3 instrucciones que comienzan en la forma `find...`:

1. `find_local_maximum(f,a,b)` (`find_local_minimum(f,a,b)`) devuelve un máximo (mínimo local) de la función f en el intervalo $[a, b]$. El máximo (mínimo) devuelto no tiene que ser el mayor (menor) de los máximos (mínimos) locales de la función en el intervalo.
2. `find_root(f,a,b)` devuelve un cero, aproximado, de la función f en el intervalo $[a, b]$. No admite precisión arbitraria ni devuelve todos los ceros. Es posible crear un programa que subdivida el intervalo en subintervalos muy pequeños y aplique la función en cada uno de ellos, en la esperanza de que haya un único cero en cada uno de los subintervalos.

7.7.5. Aproximación global

Un problema diferente, pero muy interesante, es la *aproximación global* a una función dada: supongamos que $f(x)$ sea una función continua definida en el intervalo $[0, 1]$ de la recta real. Queremos encontrar una familia de polinomios $B_{n,f}(x)$ que se “aproximen” a f en todo el intervalo $[0, 1]$ cuando n crece.

Los polinomios interpoladores, que están forzados a coincidir con la función f en un cierto número de puntos, tienen tendencia a alejarse bastante de la función en otros puntos. Entonces no resuelven el *problema de aproximación global*.

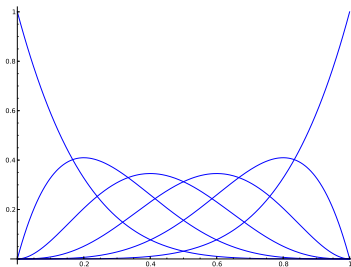
Una solución son los *polinomios de Bernstein*, definidos en la forma

$$B_{n,f}(x) := \sum_{p=0}^{p=n} \binom{n}{p} f(p/n) (1-x)^{n-p} x^p.$$

¿Cuál puede ser la idea de esta definición? Si $f(x)$ es constante e igual a 1, el polinomio $B_{n,f}(x)$ es, gracias a la fórmula del binomio de Newton, igual a $(1 - x + x)^n = 1^n = 1$. Entonces para una función constante se obtiene que todos los polinomios de Bernstein son constantes e iguales a la función.

Por otra parte, cada uno de sus sumandos $b_{n,p}(x) := \binom{n}{p}(1-x)^{n-p}x^p$ es un polinomio de grado n y todos juntos forman una base, la *base de Bernstein*, del espacio vectorial de polinomios de grado $\leq n$, de forma que $B_{n,f}(x)$ viene definido como una combinación lineal de los polinomios de la base con ciertos coeficientes que dependen de f .

Antes de continuar representamos los polinomios $b(5, p)$ mediante la instrucción `sum([plot(b(5,p),0,1) for p in xrange(6)])` que produce



y vemos que el polinomio $b(5, p)$ tiene su máximo, más o menos, en el punto $x = p/5$. Como luego estamos tomando como coeficiente de $b(5, p)$ el valor de f en p/n , podemos concluir que lo que hace la fórmula que define $B_{n,f}(x)$ es usar, cerca de p/n , la aproximación $f(p/n)b_{n,p}(x)$ y luego sumar todas esas aproximaciones. Cuando n crece usamos muchos más valores $f(p/n)$ y, se puede demostrar³, que el polinomio $B_n(x)$ se aproxima globalmente a $f(x)$.

Ejercicio 17.

1. Define una función de Sage que dependa de un entero n y una función f y devuelva $B_{n,f}(x)$.
2. Experimenta, mediante gráficas, con diversas funciones f , continuas en $[0, 1]$ y sus aproximaciones. Por ejemplo, podemos tomar $f(x) = \sin(2\pi x)$, $f(x) = \sin(4\pi x)$, etc.

³Verlas las páginas 166-167 de J.Dieudonne, *Cálculo Infinitesimal*.

3. Trata dar una definición razonable que precise en qué sentido $B_{n,f}(x)$ aproxima a $f(x)$ globalmente en el intervalo $[0, 1]$.
4. Dado que la aproximación de $B_{n,f}(x)$ a $f(x)$ es global, podría ser razonable calcular aproximadamente la integral

$$\int_0^1 f(x)dx \text{ mediante } \int_0^1 B_{n,f}(x)dx,$$

con n suficientemente grande. ¿Qué opinas de este método?

7.8. Ejercicios

Ejercicio 18.

El número e se puede obtener, entre otras, de una de las siguientes maneras

1.

$$\lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n.$$

2.

$$\sum_{n=0}^{n=\infty} \frac{1}{n!}.$$

3.

$$\lim_{n \rightarrow \infty} \frac{n^n}{(n-1)^{n-1}} - \frac{(n-1)^{n-1}}{(n-2)^{n-2}}; n > 2.$$

Este ejercicio trata de calcular valores aproximados del número e usando cada una de las expresiones anteriores y de estudiar el coste computacional de los cálculos, usando, por ejemplo. **time** o **timeit**, tratando de responder a la pregunta natural: ¿Cuál es el mejor método?

Ejercicio 19.

Consideramos la serie ⁴

$$\sum_{n=1}^{n=\infty} \frac{\text{floor}(n \cdot \tanh(\pi))}{10^n},$$

con **floor** la función *parte entera por defecto* y **tanh** la tangente hiperbólica.

COMPRUEBA que las 268 primeras cifras decimales de la suma de la serie coinciden con las primeras 268 cifras de la fracción $1/81$, pero las cifras que podemos calcular a continuación ya no coinciden.

La explicación, indicada en el libro citado en la nota a pie de página, es que la tangente hiperbólica de π está muy próxima a 1 ($0.99 < \tanh(\pi) < 1$) de forma que el numerador de los sumandos de la serie es, para muchos n igual a $n - 1$, y ahora se tiene

$$\sum_{n=1}^{n=\infty} \frac{n-1}{10^n} = \frac{1}{81}.$$

Este ejemplo ilustra claramente el peligro que corremos al suponer que algo que hemos comprobado experimentalmente, hasta el punto que nos ha permitido el *hardware* o *software* del ordenador, es una verdad matemática.

Ejercicio 20.

(APROXIMACIÓN DE STIRLING) Queremos obtener una estimación del valor del factorial de n que sea lo más precisa posible para n muy grande. Esta estimación tiene gran cantidad de usos, y es muy importante, por ejemplo, en Mecánica Estadística, que es el estudio de sistemas de partículas, gases, sólidos, etc., con un número de partículas del orden del número de Avogadro $6 \cdot 10^{23}$.

En este ejercicio combinamos ideas teóricas con cálculos en ordenador para dar una *justificación experimental* de la aproximación de Stirling.

1. Comenzamos tomando logaritmos neperianos (naturales):

$$\log(n!) = \log(1) + \log(2) + \log(3) + \cdots + \log(n),$$

y esta suma se puede calcular aproximadamente, viéndola como una suma de Riemann, como la integral

$$\int_1^n \log(x) dx = n \cdot \log(n) - n + 1.$$

⁴Propuesta en J. Borwein, D. Bailey, R. Girgensohn, *Experimentation in Mathematics*, (2004), Ed. A K Peters.

2. Nos quedamos con la aproximación $\log(n!) = n \cdot \log(n) - n$, o su equivalente exponenciando

$$n! = n^n \cdot e^{-n} = \frac{n^n}{e^n},$$

que nos dice que, en una no muy buena aproximación, el exceso que se obtiene al calcular $n^n = n \cdot n \cdot n \dots n$ veces $\dots n \cdot n$ cuando en realidad queremos $n! = 1 \cdot 2 \cdot 3 \dots (n-1) \cdot n$ se compensa, en gran medida, dividiendo entre e^n .

3. Definimos la función

$$F(n) := \frac{n!}{n^n \cdot e^{-n}},$$

y usamos Sage para calcular un gran número de valores de $F(n)$, por ejemplo $F(n)$ para n entre 1 y 50000.

4. Obtenemos una representación gráfica de los 50000 puntos obtenidos en el apartado anterior y proponemos un modelo razonable para ajustar una función a los puntos usando `find_fit`.
5. Tratamos de identificar el parámetro del modelo, es razonable en este caso, vista la gráfica, suponer que depende de un único parámetro, usando la [Enciclopedia de sucesiones de enteros](#): en la página de entrada podemos escribir las cifras decimales del valor obtenido para el parámetro, separadas por comas, y nos devuelve una lista de constantes matemáticas en las que aparece de alguna manera esa sucesión de cifras.

La Enciclopedia permite identificar sucesiones de enteros, no sólo de dígitos, como por ejemplo, la sucesión de Fibonacci y es enormemente útil al realizar *experimentos matemáticos*.

Ejercicio 21.

Define funciones $H(n, nbits)$ y $H2(n, nbits)$ que devuelvan la suma $\sum_{k=1}^{k=n} \frac{1}{k}$ calculada con precisión igual a $nbits$. Las dos funciones serán esencialmente iguales, pero H realizará todos los cálculos con números decimales de la precisión fijada mientras que $H2$ debe efectuar los cálculos con racionales, y únicamente al devolver el resultado debe convertir a decimal de $nbits$ de precisión. ¿Qué enseña este ejemplo?

Ejercicio 22.

Sabemos que la serie $\sum_{k=1}^{k=\infty} \frac{1}{k}$ es divergente. Además, sabemos que sus sumas parciales $H_n := \sum_{k=1}^{k=n} \frac{1}{k}$ tienen un valor no muy diferente a $\log(n)$ (logaritmo neperiano siempre) ya que

$$\int_1^x \frac{1}{t} dt = \log(x),$$

y H_n es una suma de Riemann de esta integral con los puntos de la partición con coordenadas enteras. Podemos entonces decir que la serie diverge (aproximadamente) como el logaritmo.

Euler definió la constante

$$\gamma := \lim_{n \rightarrow \infty} (H_n - \log(n)),$$

cuya existencia nos indica que la diferencia entre H_n y el logaritmo de n no crece con n . Sage dispone de un procedimiento, *euler_gamma*, para calcular con la precisión que queramos la constante de Euler.

1. Define una función, *mi_gamma1*($n, nbits$), que calcule el término n -ésimo de la sucesión que define *gamma* con *nbits* de precisión.
2. Define una función, *mi_gamma2*($n, nbits$), que calcule

$$\gamma_n := \sum_{k=1}^{k=n} \frac{1}{n} \left(\left\lceil \frac{n}{k} \right\rceil - \frac{n}{k} \right)$$

con *nbits* de precisión. La notación $\lceil x \rceil$ indica la parte entera por exceso de x . El límite, cuando n tiende a infinito, de γ_n se sabe que es γ .

3. Define una función, *mi_gamma3*($n, nbits$), que calcule *gamma* sumando n términos de la serie

$$\sum_{k=1}^{k=\infty} \left(\frac{1}{k} - \log\left(1 + \frac{1}{k}\right) \right),$$

que también converge a γ .

4. Compara la cantidad de cifras correctas obtenidas y los tiempos de cálculo para los tres métodos usando $n = 10^6$ y fijando la precisión óptima mediante experimentación. ¿Qué conclusiones obtienes y cuál puede ser el motivo?

Ejercicio 23.

Se define la función $\sigma(n)$ como la suma de todos los divisores positivos del entero positivo n . Sage dispone de la función *sigma* que realiza este cálculo. Determina (experimentalmente) el mayor entero positivo tal que la diferencia

$$e^\gamma \cdot n \cdot \log(\log(n)) - \sigma(n),$$

es negativa (γ es la constante de Euler).

Ejercicio 24.

Definimos una sucesión de números racionales mediante

$$G_1 = 0, G_2 = 1, G_n = G_{n-1} + \frac{1}{n-2} G_{n-2} \quad (n \geq 3).$$

1. Comprueba (experimentalmente) que la fracción $\frac{n}{G_n}$ tiende al número e .
2. Compara la eficiencia de esta manera de aproximar e con la de la forma, más conocida, que utiliza la serie

$$e = \sum_{n=0}^{n=\infty} \frac{1}{n!}.$$

Ejercicio 25.

Podemos estimar el valor de $\alpha(x) := \sqrt{1+x}$ como $\beta(x) := 1 + (x/2)$ cuando x es, en valor absoluto, suficientemente próximo a cero. Consideramos que la estimación es aceptable cuando el error relativo

$$E(x) := \frac{\text{abs}(\alpha(x) - \beta(x))}{\alpha(x)}$$

que cometemos es menor que 0.01.

1. Determina, experimentalmente, una cota $|x| < A$ tal que la estimación es aceptable para los x que la cumplen, y deja de serlo para $|x| \geq A$.
2. ¿Qué podemos hacer para mejorar la estimación? Postula una nueva estimación $\beta_1(x)$ y demuestra que con ella se obtienen resultados bastante más precisos.

Ejercicio 26.

John Napier publicó su invención de los logaritmos en 1614, mucho antes de la invención del cálculo diferencial. El punto esencial para que los logaritmos fueran útiles era la existencia de **tablas de logaritmos** en las que uno encontraba los logaritmos de los factores, que sumaba a mano, y volvía a usar para encontrar el número cuyo logaritmo era la suma obtenida, es decir, el producto de los números de partida. Henry Briggs colaboró con Napier para producir las primeras tablas de logaritmos mediante el siguiente procedimiento:

1. Queremos calcular el logaritmo de un entero $n > 1$. Calculamos $n^{1/2^K}$ para $K = 1, 2, 3, \dots$ hasta que el resultado difiera muy poco de 1. Esto es lo mismo que iterar la extracción de raíces cuadradas hasta llegar casi a 1. Escribimos

$$n^{1/2^K} = 1 + x.$$

2. Ahora tomamos logaritmos para obtener

$$\log(n) = 2^K \log(1 + x).$$

3. Finalmente, Briggs observó que para x suficientemente pequeño en valor absoluto, podía sustituir $\log(1 + x)$ por x , y quedaba $\log(n)$ aproximadamente igual a $2^K x$.

Es claro que lo que necesitamos es saber **cómo de pequeño tiene que ser x para obtener un número prefijado k de cifras decimales exactas del logaritmo de n .**

1. Define una función `buscar(n, k, precis)` que debe efectuar el procedimiento de Briggs hasta que la estimación obtenida tenga el número k de cifras correctas de $\log(n)$, y en ese momento debe devolver x . La precisión arbitraria en los cálculos hace falta porque sin ella un cierto bucle **while** puede hacerse infinito cuando n o k son grandes.
2. Experimenta con la función del primer apartado y, como consecuencia, enuncia y comprueba la regla que indica cómo de pequeño tiene que ser x para obtener k cifras correctas del logaritmo de n .

Capítulo 8

Criptografía

La Criptografía, o más en general la Teoría de códigos (compresores, correctores, criptopgráficos), es una de las aplicaciones más importantes de las Matemáticas al mundo real. En el caso de la Criptografía se trata, en gran parte, de una aplicación de la teoría de números, una parte de las Matemáticas que antes de llegar esta aplicación se consideraba *absolutamente pura*.

Para encriptar¹ un mensaje utilizamos una *clave*, perteneciente a un cierto conjunto \mathcal{K}_c , y un *algoritmo* que, para cada mensaje y cada clave, nos da un mensaje encriptado. Además, para cada clave de encriptado necesitamos una clave de desencriptado, perteneciente a un cierto conjunto \mathcal{K}_d , de forma que ambos conjuntos de claves deberían ser biyectivos mediante una biyección explícitamente conocida. En el capítulo 10 se recuerdan algunas nociones básicas, que deben ser conocidas gracias a la asignatura CONJUNTOS Y NÚMEROS, sobre los diversos tipos de funciones.

Por supuesto, necesitamos también un algoritmo de desencriptado, que para cada mensaje encriptado y cada clave de desencriptado devuelva el mensaje original. En muchos casos el algoritmo de desencriptado es el mismo que el de encriptado.

La seguridad de un sistema criptográfico no debe residir en mantener secreto el algoritmo de encriptado, ya que si un método de encriptado se le ocurre a una persona también se le puede ocurrir a otra, sino en claves, que en parte o totalmente deben mantenerse secretas, pertenecientes a un espacio de claves enorme, cuanto más grande más seguro es el sistema, y elegidas aleatoriamente, veremos en el capítulo 11 cómo hacerlo, dentro de él.

¹Usamos como equivalentes los términos *encriptar* y *cifrar*, *desencriptar* y *descifrar*.

8.1. Codificación

Un texto, como los que queremos cifrar, es una cadena de caracteres y las funciones Φ y Ξ , que cifran y descifran, deberían ser definidas mediante operaciones matemáticas, es decir mediante ciertas *fórmulas*, ya que operando directamente sobre las cadenas de caracteres sólo podemos realizar, de forma cómoda, operaciones muy simples, como *cambiar la A por la M, la B por la Z, etc.*

Entonces, es necesario para definir de manera sencilla y sistemática las funciones del sistema criptográfico, cambiar las cadenas de caracteres por números, por ejemplo enteros, o quizá por cadenas de bits. A esta fase previa al cifrado se le llama CODIFICACIÓN.

La codificación puede verse como una función biyectiva ϕ del conjunto de mensajes \mathcal{M} en un cierto conjunto “numérico”, por ejemplo podría ser el conjunto de enteros módulo un N muy grande \mathbb{Z}_N . Denotamos mediante \mathcal{N} y \mathcal{NC} estos “conjuntos de números”, el de los mensajes antes de encriptar y el de los mensajes encriptados.

Entonces, la función Φ es la composición de la función que codifica ϕ , multiplicada por la identidad en las claves, con la verdadera función que encripta Γ , definida mediante procesos matemáticos, y, si queremos enviar un mensaje de texto, la composición con una tercera función φ , también biyectiva, que DESCODIFICA, es decir, transforma el “número” que obtuvimos al encriptar en una cadena de caracteres:

$$\begin{array}{ccc} \mathcal{M} \times \mathcal{K}_c & \xrightarrow{\Phi} & \mathcal{MC} \\ \downarrow \phi \times 1 & & \uparrow \varphi \\ \mathcal{N} \times \mathcal{K}_c & \xrightarrow{\Gamma} & \mathcal{NC} \end{array}$$

La función Ξ también puede escribirse como una composición de una función que codifica, multiplicada por la identidad en el conjunto de claves \mathcal{K}_d , con la función que descifra y con la que descodifica. Es claro que un sistema criptográfico es de utilidad si para cualquiera mensaje que encriptemos usando una cierta clave de encriptado podemos descifrarlo usando la correspondiente clave de descifrado. En todos los ejemplos sobre los que trabajaremos tendremos que comprobar que se cumple esta condición al menos para un mensaje suficientemente largo.

La codificación depende de la forma en que vayamos a cifrar, y por tanto puede variar de un sistema criptográfico a otro. Debemos entender que CODIFICAR un texto, de forma que, por ejemplo, lo transformamos en una cadena de ceros y unos, NO es lo mismo que CIFRARLO: el texto codificado puede ser ilegible en principio, pero basta tener una idea del sistema de codificación que se usó para poder leerlo. En particular, la codificación no utiliza claves.

Puedes ver varios ejemplos de codificaciones en la hoja de Sage

[81-CRIPT-codificacion.ipynb](#).

8.2. Criptoanálisis

Antes de pasar a describir en detalle algunos sistemas criptográficos conviene describir dos métodos básicos del llamado *criptoanálisis*, es decir, el estudio de los métodos para romper sistemas criptográficos.

1. Como el conjunto de posibles claves es finito, es, en principio, posible ir probando claves hasta obtener un texto legible. Se llama a esto “un ataque mediante fuerza bruta”. Podemos hacer muy difícil, imposible, un ataque de fuerza bruta haciendo enorme el espacio de claves. Hay que tener en cuenta que un sistema seguro puede dejar de serlo si se dan grandes mejoras en el *hardware* o el *software* utilizados para romperlo.
2. Algunos sistemas criptográficos clásicos podían ser atacados mediante análisis estadísticos del texto encriptado, estudiando las frecuencias de las letras, de los pares de letras, etc.

El texto encriptado puede, para algunos sistemas criptográficos primitivos, contener todavía alguna información acerca del texto legible que puede ser extraída.

8.3. Criptografía clásica

En esta sección describiremos algunos sistemas clásicos de cifrado que cubren en parte la historia de la Criptografía hasta el comienzo del siglo XX.

8.3.1. Cifra de César

Es quizá el sistema criptográfico más antiguo, y más simple, que se conoce:

1. Cada una de las 26 letras se codifica como un entero módulo 26. Esta asignación puede ser arbitraria, pero normalmente la hacemos en orden: la ‘A’ al cero, la ‘B’ al uno, etc. Un mensaje se codifica codificando sus letras en orden.
2. El conjunto de claves para encriptar \mathcal{K}_c y el de claves para desencriptar \mathcal{K}_c son el mismo, e igual a los enteros módulo 26.

3. El conjunto \mathcal{N} es $\bigcup_{1 \leq n \leq N} \mathbb{Z}_{26}^n$, con N la longitud máxima de los mensajes.
4. Supongamos elegida una clave $k \in \mathbb{Z}_{26}$. El encriptado se hace letra a letra y consiste simplemente en sumar, siempre módulo 26, la codificación de la letra con la clave.
5. Se descodifica, letra a letra por ejemplo, con la inversa de la función que codifica.
6. La clave para descryptar un mensaje encriptado con la clave k es $26 - k$.

Es muy fácil romper el sistema de César probando las posibles claves, y parando cuando se encuentre un texto legible. Aparte de este ataque de *fuerza bruta*, el sistema de César también puede ser atacado fácilmente, siempre que dispongamos de un texto encriptado suficientemente grande, mediante análisis de frecuencias de las letras en el texto encriptado y comparación con las frecuencias de las letras en la lengua del mensaje.

Veremos ejemplos de todo esto en la hoja de Sage [82-CRIPT-cesar.ipynb](#).

8.3.2. Cifrado de permutación

La función “sumar k ” de \mathbb{Z}_{26} en sí mismo es una biyección, y la debilidad del sistema de César reside en que únicamente hay 26 funciones de esta clase, y una de ellas no sirve. Se puede mejorar algo la seguridad del sistema usando una biyección arbitraria de \mathbb{Z}_{26} en sí mismo como clave, ya que hay

$$26! = 403291461126605635584000000$$

biyecciones de \mathbb{Z}_{26} en sí mismo. Elegida una tal biyección ϕ , que podemos pensar como una biyección del alfabeto \mathcal{A} en sí mismo (no hace falta codificación), simplemente cambiamos cada letra $*$ del mensaje por $\phi(*)$ para obtener el mensaje encriptado. Por supuesto, para descryptar usamos la biyección inversa.

No es imposible un ataque de fuerza bruta, pero ya sería más costoso. Sin embargo, este sistema puede ser atacado mediante análisis de frecuencias porque cada letra se encripta siempre mediante la misma letra, de forma que la letra más frecuente en la lengua del mensaje original se encripta siempre por la misma, que pasa a ser la más frecuente del mensaje encriptado, etc.

En la época en que se usaban estos sistemas criptográficos, el análisis de frecuencias permitía descifrar algunos de los caracteres, pero también se usaba la habilidad para rellenar crucigramas del criptoanalista para completar la información que suministraba el análisis de frecuencias.

8.3.2.1. Oscurecimiento de las frecuencias

Supongamos un sistema como el cifrado de permutación, fácilmente atacable mediante análisis de frecuencias. Podemos “oscurecer” las frecuencias de las letras mediante el siguiente truco: en lugar de 26 caracteres usamos, por ejemplo, un alfabeto de 100 símbolos, y a una letra, por ejemplo E , cuya frecuencia en la lengua que usamos es, por ejemplo, 12 % le asignamos 12 símbolos, eligiendo al azar uno entre los doce cada vez que nos encontramos una E .

De esta forma conseguimos que cada uno de los 100 símbolos de nuestro alfabeto extendido tenga una frecuencia de un 1 % y el texto encriptado ya no tiene información relevante acerca de la frecuencia de cada uno de sus símbolos. Aún así, todavía queda información relevante, y explotable, acerca de los pares, y tripletas, de letras.

8.3.3. Cifra de Vigenere

Para mejorar el sistema de César, de Vigenere inventó un sistema en el que se utiliza una clave de César distinta según la posición que ocupa el caracter en el texto.

Una manera simple de describirlo es

1. Elegimos una palabra de cierta longitud que será la clave, por ejemplo ‘CIRUELA’.
2. Pensamos que el texto está todo en una sola línea y escribimos debajo la palabra clave repetida sin espacios hasta que sobrepasamos la longitud del texto.
3. Cada letra del texto se encripta mediante César usando como clave la posición en el alfabeto de la letra de la palabra clave que tiene debajo.

Si usamos palabras clave de longitud k , el espacio de claves tiene 26^k posibles claves. Es razonable, aunque esto hace más difícil recordarla, que la palabra clave no sea una existente. Haciendo k suficientemente grande, podemos conseguir espacios de claves enormes a costa de complicar el mantenimiento o transmisión de las claves.

Como veremos en las hoja de Sage [83-CRIPT-vigenere.ipynb](#) el sistema de Vigenere puede ser atacado, mediante análisis de frecuencias, si disponemos de suficiente texto encriptado, pero cuanto mayor sea la longitud de la clave necesitaremosmos más cantidad de texto encriptado.

8.3.4. Cifrado matricial

Otra variante del sistema de César consiste en usar *biyecciones afines* de \mathbb{Z}_{26} en sí mismo, es decir funciones $f(x) := a \cdot x + b$, con a invertible y b arbitrario en \mathbb{Z}_{26} . Esto no mejora casi la seguridad del sistema de César, ya que sigue habiendo muy pocas biyecciones afines, pero la idea se puede generalizar para resolver este problema.

Basta usar transformaciones afines $f(\mathbf{X}) := \mathbf{A} \cdot \mathbf{X} + \mathbf{b}$ con \mathbf{A} una matriz $n \times n$ invertible con entradas en \mathbb{Z}_{26} , y \mathbf{X} y \mathbf{b} vectores en \mathbb{Z}_{26}^n . Es cierto que \mathbb{Z}_{26} no es un cuerpo, de forma que \mathbb{Z}_{26}^n no es un espacio vectorial, pero sigue siendo válido que una matriz \mathbf{A} con entradas en \mathbb{Z}_{26}^n es invertible si y sólo si su determinante es distinto de cero.

1. Los mensajes se codifican letra a letra usando el mismo sistema que en César.
2. Cada mensaje se divide en trozos, bloques, de n caracteres, y cada trozo queda codificado mediante un vector en \mathbb{Z}_{26}^n . Si el último trozo no llega a n caracteres se añaden al final las zetas que hagan falta.
3. Se encripta usando la función afín biyectiva $f(\mathbf{X}) := \mathbf{A} \cdot \mathbf{X} + \mathbf{b}$, y, en consecuencia, se desencripta mediante $g(\mathbf{Y}) := \mathbf{A}^{-1} \cdot (\mathbf{Y} - \mathbf{b})$, con todas las operaciones realizadas módulo 26.

Tomando n suficientemente grande se consigue que el espacio de posibles claves sea muy grande, y además, como cada letra se encripta de distinta manera dependiendo de las otras letras que están en su mismo bloque, las frecuencias de las letras quedan muy alteradas y no parece útil un análisis de frecuencias.

Sin embargo, si conseguimos $n + 1$ vectores en \mathbb{Z}_{26}^n y los $n + 1$ vectores que les corresponden mediante la transformación afín, y siempre que los $n + 1$ vectores formen un sistema de referencia afín (i.e. restando uno de ellos a los restantes n se obtiene un conjunto linealmente independiente), se puede obtener la transformación afín, es decir la clave².

En la hoja de Sage [84-CRIPT-matricial-fuerzabruta.ipynb](#) puedes encontrar un ejemplo de uso de este método, usando matrices 2×2 , junto con un ataque mediante *fuerza bruta*, posible porque el espacio de claves no es muy grande, usando un diccionario.

Podríamos pensar en hacer más seguro este sistema usando una clave distinta para encriptar cada bloque, cada vector, pero esto complica muchísimo la logística de las claves y es más simple, e igual de seguro, usar el sistema tratado en la siguiente subsección.

²La información necesaria, texto sin encriptar y el correspondiente texto encriptado, se podría obtener mediante agentes dobles, que, al menos en las películas, siempre existen.

8.3.5. One time pad

Este es un sistema muy simple y para el que se puede demostrar matemáticamente que, siempre que los malos³ no nos roben las claves, es absolutamente seguro.

1. En este sistema criptográfico LA CLAVE, la misma para encriptar y desencriptar, es una cadena inmensa de ceros y unos generada aleatoriamente. Por ejemplo, podemos suponer que la clave se guarda en un DVD de 4.7GB (Gigabytes) y, por tanto, consiste en $4.7 \times 8 \times 10^9$ ceros o unos. Al afirmar que la clave se ha generado aleatoriamente queremos decir que la probabilidad de un uno es la misma que la de un cero, de forma que en cada subcadena suficientemente larga aproximadamente la mitad serán ceros, y el resto unos.

El emisor y el receptor del mensaje tienen una copia de la clave, el DVD, que, por supuesto, deben mantener en secreto.

2. Para encriptar un mensaje debemos

- a) CODIFICARLO EN BINARIO: Suponemos que el mensaje está formado por una única cadena, sin espacios, usando 26 letras mayúsculas. Como $26 < 2^5 = 32$ podemos usar una palabra binaria de 5 bits para representar cada letra del alfabeto.

Si el mensaje tenía N letras queda codificado mediante una cadena binaria de $5 \times N$ bits ($5 \times N$ ceros o unos).

- b) ENCRIPtarlo: tomamos los primeros $5 \times N$ bits de la clave y encriptamos cada bit del mensaje sumándole el bit que ocupa el mismo lugar en la clave mediante la regla

$$0 + 0 = 0; 0 + 1 = 1 = 1 + 0; 1 + 1 = 0.$$

LE QUITAMOS A LA CLAVE, el emisor y el receptor, los $5 \times N$ bits ya usados. El siguiente mensaje que se encripte usará entonces bits de la clave original a partir del que ocupa el lugar $5 \times N$.

- c) DESCODIFICARLO: Hemos obtenido una cadena binaria de longitud $5 \times N$ que podemos descodificar para obtener un texto que enviaríamos. Observa que para codificar sólo necesitamos conocer la palabra binaria que codifica cada letra, pero para poder descodificar necesitamos usar un alfabeto de 32 caracteres, de forma que a toda palabra binaria de 5 bits le corresponda, de forma biunívoca, uno de los caracteres del alfabeto.

³Es claro que, seamos quienes seamos, los malos siempre son los otros.

Supondremos entonces que el alfabeto es

$$\mathcal{A} = ' ABCDEFGHIJKLMNOPQRSTUVWXYZ ;, , #\$@'$$

Ejercicio 1.

DEBES operar en cada momento con listas o cadenas de caracteres según te convenga, y, por tanto, puedes necesitar conversiones entre listas y cadenas de caracteres.

1. Genera una clave de longitud 10^6 (no debes mostrar por pantalla el resultado). Divide la clave en 10 trozos de la misma longitud y determina, para cada trozo, la diferencia entre el número de ceros que hay en el trozo y el número que debería haber.
2. Escribe funciones para codificar y decodificar un texto, en el alfabeto ampliado de 32 caracteres \mathcal{A} . La primera debe devolver, si N es el número de letras del texto, una cadena binaria de longitud $5 \times N$ y la segunda debe ser la inversa de la primera. Comprueba, sobre el texto dado en la hoja de SAGE que se adjunta, que las funciones son inversas una de la otra.
3. Escribe una función para encriptar un texto ya codificado en binario. ¿Cómo se desencripta un texto encriptado mediante este sistema? Encripta el texto propuesto en la hoja de SAGE que se adjunta, obteniendo un texto en el alfabeto \mathcal{A} , y desencriptalo para recuperar el texto original.
4. ¿Por qué es importante no reutilizar trozos de la clave ya usados? Indica tu opinión RAZONADA sobre la seguridad de este sistema criptográfico.

En particular, ¿podría tener éxito un ataque “mediante fuerza bruta” con diccionario?

5. El modo como se encripta se puede describir como “suma en binario sin llevar”. Podríamos pensar en cambiar el sistema para efectuar las sumas “llevando”, de forma que $1 + 1$ deja un cero en la columna en la que estamos pero añade un uno a la siguiente columna.

¿Le ves algún problema a esta modificación del sistema original? EN ESTE EJERCICIO DEBES EMPEZAR resolviendo algunos ejemplos pequeños en papel, y si tienes claro que el método funciona correctamente puedes tratar de implementarlo mediante una función que encripte y una que desencripte.

Puedes ver una solución en la hoja de Sage [88-CRIPT-one-time-pad.ipynb](#).

8.3.6. En resumen

La evolución en el tiempo de los sistemas criptográficos clásicos les ha llevado a

1. Incrementar el tamaño del conjunto de claves para hacer inviable un ataque de “fuerza bruta”. Un ejemplo es el paso de la cifra de César a la de Vigenere.
2. Evitar un ataque mediante análisis de frecuencias, tratando de no encriptar las mismas letras de la misma manera. En el sistema de Vigenere, si la longitud de la clave era k , teníamos k maneras de encriptar cada letra, pero vimos que dividiendo el texto en k trozos era posible, siempre que dispusiéramos de suficiente texto encriptado, aplicar un análisis de frecuencias y romperlo.

El remedio a este problema es hacer k al menos tan grande como la longitud del texto a encriptar, y ese es el origen del “*One time pad*”, que es un sistema que podemos demostrar que, si se usa bien, es totalmente seguro.

8.4. Clave pública: RSA

La idea de la criptografía de clave pública es sencilla: cada usuario del sistema debe tener dos claves: una clave pública y una clave privada. La clave pública se usa para encriptar un mensaje dirigido a ese usuario, mientras que la clave privada, que sólo él debe conocer, se usa para descifrar un mensaje que se ha cifrado usando la clave pública del usuario.

Por supuesto, las dos claves no serán independientes, pero debe estar garantizado, hasta donde sea posible, que el conocimiento de la clave pública NO permita obtener la clave privada.

1. El sistema de encriptación RSA es de clave pública: se encripta un mensaje para el usuario A usando la clave pública de A , y A desencripta el mensaje usando su clave privada. Cada usuario debe tener una clave pública y una privada.
2. La seguridad de RSA radica en que la clave pública de cada usuario depende del producto de dos números primos muy grandes, diferentes para cada usuario, mientras que la clave privada de cada usuario depende de los dos primos cuyo producto forma parte de su clave pública.

Si los primos son muy grandes su producto lo es más, y basta que ese producto sea mucho más grande que los mayores enteros que podemos factorizar con la tecnología actual para que, de momento, RSA sea seguro.

3. Lo primero es codificar el mensaje como un entero al que podemos aplicar operaciones aritméticas para encriptarlo. Supongamos que queremos codificar mensajes de longitud N escritos en un alfabeto de 32 letras, las del alfabeto castellano con signos de puntuación y “espacio”. Mensajes posibles hay 32^N y podemos representar cada uno de ellos, \mathcal{M} , como un entero $C(\mathcal{M})$ de N cifras en base de numeración 32: Para cada letra del mensaje, por ejemplo la letra i -ésima suponiendo que es la j -ésima del alfabeto, añadimos el sumando $j \cdot 32^i$ al entero $C(\mathcal{M})$.

4. El entero más grande que podemos escribir así es

$$M_N := 31 + 31 \cdot 32 + 31 \cdot 32^2 + \cdots + 31 \cdot 32^{N-1}.$$

Esta correspondencia entre mensajes, no tienen que tener significado, de longitud N y enteros entre 0 y M_N es biyectiva. Esta correspondencia codifica, que no encripta, los mensajes.

5. Supongamos que A elige como parte de su clave pública un entero n que ha construido multiplicando dos primos muy grandes. Necesitamos que M_N sea menor que n de forma que podemos ver cada mensaje $m := C(\mathcal{M})$ como un elemento $[m]$ de \mathbb{Z}_n (un resto $[m]$ módulo n).
6. Dado n y sus factores primos p y q , calculamos el número $\phi(n) = (p-1) \cdot (q-1)$ de clases de restos invertibles módulo n , con ϕ la “función de Euler”. Una propiedad importante de ϕ , el teorema de Fermat-Euler, es que, para todo elemento $[m]$ en \mathbb{Z}_n tal que m es primo con n , se verifica $[m]^{\phi(n)} = [1]$.
7. La CLAVE PÚBLICA de A está formada por n y un entero e tal que su clase $[e]$ módulo $\phi(n)$ sea invertible, es decir, tal que existe un entero d verificando $e \cdot d = 1 + k \cdot \phi(n)$ para un cierto entero k . Sabemos que tales enteros e son los que son primos con $\phi(n)$.
8. La CLAVE PRIVADA de A consiste en el entero d , que debe mantenerse secreto. El usuario A conoce los primos p y q , porque ha generado n multiplicándolos, y elegido e puede calcular fácilmente d usando el teorema de Bezout.
9. Denotemos por m el entero que codifica el mensaje \mathcal{M} , es decir $m := C(\mathcal{M})$.

Podemos suponer que el mensaje codificado m es primo con n , porque si no lo fuera siempre podríamos añadir espacios, sin estropear el mensaje, hasta conseguir un mensaje con codificación un entero primo con n .

10. Para ENCRYPTAR \mathcal{M} empezamos calculando el entero $m := C(\mathcal{M})$, y simplemente elevamos $[m]$ al exponente e en \mathbb{Z}_n . Para encriptar un mensaje con destino a A necesitamos la clave pública (n, e) de A completa.

11. El resultado $[m]^e$ puede que no esté en el conjunto de clases de restos que son imagen de un mensaje de N letras, pero podemos ajustar N de forma que entre M_N y M_{N+1} haya suficientes claves públicas para todos los usuarios potenciales del sistema.

Entonces, podemos volver a escribir $[m]^e$ como un mensaje de texto, pero uno de $N + 1$ letras. Este último mensaje es el que enviamos.

12. DESENCRIPTADO: En primer lugar representamos el mensaje recibido como un entero de $N + 1$ cifras en base 32. Si conocemos n , $[m]^e$ y d podemos recuperar $[m]$ mediante

$$([m]^e)^d = [m]^{e \cdot d} = [m]^{1+k \cdot \phi(n)} = [m] \cdot ([m]^{\phi(n)})^k = [m] \cdot [1] = [m].$$

Este cálculo prueba que el encriptado de mensajes m , tales que m es primo con n , es inyectivo.

13. Para terminar hay que decodificar m , es decir, hay que transformar m en un mensaje de texto de N letras, escribiendo m en base 32 y cambiando las cifras resultantes por letras del alfabeto.
14. Aunque se conozcan n y e , sin factorizar n no es posible calcular $\phi(n)$, de forma que no se puede calcular el inverso, d , de e módulo $\phi(n)$ que hace falta para desenscriptar.
15. En la práctica, los primos p y q deben ser muy grandes, por ejemplo más de 200 cifras, y deben cumplir algunas condiciones adicionales como, por ejemplo, no estar próximos.
16. Encriptar mensajes largos usando RSA con claves seguras, por ejemplo claves de 2048 bits, es bastante costoso en tiempo de ordenador. Entonces, RSA no sirve para encriptar la comunicación, mediante páginas web de, por ejemplo, un cliente con su banco.

Lo que se hace, por ejemplo cuando la dirección a la que conectamos empieza por <https://>, es usar RSA para intercambiar, de manera segura, claves de otro sistema de encriptado, típicamente basado directamente en operaciones binarias, más rápido pero menos seguro. Esas claves, que no son claves públicas, sólo se utilizan una vez, de forma que no se compromete mucho la seguridad.

Puedes ver una solución de este ejercicio en la hoja de Sage [Puedes ver la solución en el archivo de Sage 85-CRIPT-rsa.ipynb](#).

8.5. Otro sistema de clave pública: MH

1. Una sucesión, finita o infinita, de números enteros positivos a_1, a_2, a_3, \dots se dice que es “supercreciente” si para todo $n > 1$ verifica $a_n > \sum_{i < n} a_i$.
2. Define una función de SAGE que compruebe si una sucesión finita de enteros positivos dada es supercreciente o no. La función debe tener como argumento la lista de enteros de la sucesión $[a_1, a_2, \dots, a_n]$.
3. Define otra función que genere una sucesión supercreciente de longitud n de acuerdo a la siguiente regla: fijamos otro entero positivo N y elegimos el término a_j de la sucesión aleatoriamente en el intervalo $(\sum_{i < j} a_i, \sum_{i < j} a_i + N]$ (el término a_1 estará en el intervalo $[1, N]$). Esta segunda función nos interesa para disponer de suficientes ejemplos de sucesiones supercrecientes sin tener que generarlos a mano. Comprueba que algunos de los ejemplos generados por la segunda función son supercrecientes según la primera.
4. Dados una sucesión supercreciente $[a_1, a_2, \dots, a_n]$ y un entero positivo A queremos saber si A es la suma de algunos de los enteros de la sucesión, es decir, si existe una solución de la ecuación

$$A = \sum_1^n x_i \cdot a_i \quad (8.1)$$

con todos los x_i iguales a 0 o a 1.

5. En primer lugar implementamos un método de “fuerza bruta”, para lo que comenzamos generando la lista de todas las listas de ceros y unos de longitud n :

```
def listas(K):
    L = []
    for k in srange(2^K):
        L.append(k.digits(base=2, padto=K))
    return L
```

Usando esta función, define una, con parámetros la lista de términos de una sucesión supercreciente y el entero A , que busque, recorriendo la lista de listas, y devuelva, si la encuentra, una solución de (8.1). ¿Siempre hay solución?

6. Finalmente, busquemos una solución mediante un algoritmo más eficiente, que debes implementar:

- a) Si $A \geq a_n$ entonces x_n debe ser uno, y en caso contrario debe ser cero (¿por qué?).
- b) Para calcular x_{n-1} repite el argumento cambiando A por $A - x_n \cdot a_n$ y $[a_1, a_2, \dots, a_n]$ por $[a_1, a_2, \dots, a_{n-1}]$.
- c) Repitiendo podemos calcular todos los x_i , y si existe una solución el algoritmo la encuentra y es única. Debe ser claro que este algoritmo sólo funciona para sucesiones supercrecientes.

A continuación se explican los detalles de un método de criptografía de clave pública, debido a Merkle y Hellman, y basado en la dificultad de resolver, por fuerza bruta ya que no se conoce otro método, una ecuación como (8.1) cuando la sucesión de coeficientes no es supercreciente y el número de incógnitas es muy grande.

- 7. Cada usuario elige una sucesión supercreciente $[a_1, a_2, \dots, a_N]$, de la misma longitud N para todos, un entero $m > 2a_N$ y otro entero w primo con m . Estos datos son su clave privada.
- 8. Cada usuario calcula la sucesión $[b_1, b_2, \dots, b_N]$ haciendo $b_i := w \cdot a_i \% m$ y esa nueva sucesión será su clave pública.
- 9. Cada letra de un mensaje a encriptar se sustituye por la representación binaria del correspondiente entero entre 0 y 25.
- 10. El mensaje es ahora una cadena de ceros y unos que se divide en bloques de longitud N . Para cada uno de esos bloques x_1, x_2, \dots, x_N se encripta mediante

$$c := \sum_1^N b_i \cdot x_i$$

de forma que el mensaje encriptado es una lista de números enteros.

- 11. Para desencriptar el bloque c se usa el inverso \bar{w} de w módulo m y se calcula

$$v := \bar{w} \cdot c = \sum \bar{w} \cdot b_i \cdot x_i \equiv \sum a_i \cdot x_i \pmod{m}.$$

pero, como la sucesión de los a_i es supercreciente, se verifica que la suma de todos los a_i es menor que $2 \cdot a_N$ y, por tanto menor que m . Entonces v , no sólo es congruente con $\sum a_i \cdot x_i$ módulo m sino que, es igual como número entero a $\sum a_i \cdot x_i$.

12. Para terminar de descryptar basta usar, ya que la sucesión es supercreciente, el algoritmo que aparecía en el punto 6 de esta misma sección.

Se ha demostrado (Shamir, 1982) que este método no es totalmente seguro, y se han propuesto algunas variantes que, de momento, se consideran seguras.

Ejercicio 2. Programar el encriptado y descryptado usando este método. Puedes ver una solución en la hoja de Sage [87-CRIPT-mhellman.ipynb](#).

8.6. Firma digital

1. Un sistema de FIRMA DIGITAL debe servir para que el receptor de un mensaje, por ejemplo de correo electrónico, pueda comprobar que
 - a) el mensaje no ha sido alterado durante su transmisión. El correo electrónico viaja, desde el ordenador del emisor hasta el del receptor, a través de un cierto número, que puede ser grande, de servidores de correo electrónico intermedios.
 - b) que el mensaje realmente procede de la persona que afirma que lo ha enviado. No es difícil enviar mensajes de correo electrónico en los que aparece como dirección de correo electrónico del emisor una dirección que no es nuestra. Se llama a esta práctica “*email spoofing*”.
2. Mediante un sistema criptográfico de clave pública, como RSA, es fácil implementar un sistema de firma digital.
3. Si queremos firmar un mensaje \mathcal{M} lo primero que hacemos es calcular un “número resumen” (*hash*) del mensaje, que hemos escrito como un archivo de texto. Por ejemplo, en una terminal de Linux podemos usar un comando como `md5sum <nombre del archivo>` y obtenemos un número hexadecimal de 32 cifras que “resume” el mensaje.

No es posible que la función que asocia a cada archivo su hash sea inyectiva ya que el número de mensajes, de tamaño razonable, posibles es muchísimo mayor que 16^{32} . Sin embargo, la función se diseña de forma que sea prácticamente imposible generar un mensaje con significado, y con el significado que queremos, y con el mismo hash que un mensaje dado.

4. Encriptamos el hash del mensaje CON NUESTRA CLAVE PRIVADA y añadimos el hash encriptado al final del mensaje, que podemos ya enviar, encriptándolo para su destinatario o no.
5. El receptor debe desencriptar el mensaje con su clave privada, si estaba encriptado, desencriptar el hash que estaba al final usando la CLAVE PÚBLICA DEL EMISOR y comprobar que el hash del mensaje recibido, después de quitarle la última línea, coincide con el hash que hemos obtenido al desencriptar la última línea.
6. Si coincide vemos que la única manera en que el mensaje puede haber sido falsificado es por alguien que conoce la clave privada del supuesto emisor. Queda claro entonces que alguien que conozca nuestra clave privada no sólo puede desencriptar nuestros mensajes sino que además puede suplantarnos en la red.

Ejercicio 3. *Enviar, a la dirección de correo del profesor, un mensaje firmado que contenga dos líneas:*

1. *La primera contiene el número n de tu clave pública seguido de una G y el exponente e de tu clave pública seguido de otra G .*
2. *La segunda línea contiene el hash `md5sum` del archivo cuyo único contenido es la línea descrita en el punto anterior encriptado usando tu clave privada. Es importante que el archivo no debe tener líneas en blanco a continuación de la línea prescrita, y debe ser un archivo de puro texto, generado por ejemplo con `Kwrite` o `emacs`.*
3. *No es necesario encriptar el mensaje de correo que se envía, y, por tanto no se indica la clave pública del profesor.*

El sistema criptográfico que vamos a usar, RSA con un alfabeto de 16 caracteres, va a permitir enviar hasta 64 caracteres de un alfabeto formado por los símbolos

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$

es decir, los dígitos hexadecimales. En consecuencia, el entero n de las claves públicas debe estar comprendido entre 16^{64} y 16^{65} .

8.7. Ejercicios

Ejercicio 4.

Sea p un entero primo, y consideramos el conjunto \mathbb{Z}_p^* de clases de restos módulo p , distintas de la clase nula, con la operación de multiplicación. Una forma de encriptar mensajes, bastante parecida a RSA, consiste en representar el mensaje como un entero M en \mathbb{Z}_p^* , con p muy grande y, elegido un $e \in \mathbb{Z}_p^*$ tal que $\text{MCD}(e, p-1) = 1$, encriptar el mensaje como el elemento M^e de \mathbb{Z}_p^* . Para desencriptar hace falta calcular el inverso multiplicativo d de e módulo $p-1$, es decir $d \cdot e = 1 + k \cdot (p-1)$, ya que, gracias al teorema pequeño de Fermat, $(M^e)^d = M$ en \mathbb{Z}_p^* . El entero e es la clave de cifrado y d es la de descifrado.

En este ejercicio queremos ver que si un atacante consigue conocer un mensaje M y su correspondiente mensaje cifrado $MC := M^e$ puede extraer mucha información sobre la clave e con cierta facilidad. A lo largo del ejercicio hay que efectuar un montón de veces operaciones del tipo M^e módulo p , que deben hacerse en la forma más eficiente posible para que la cosa marche razonablemente bien (ver la función `power_mod` de Sage o la sección ***).

1. El orden de un elemento $M \in \mathbb{Z}_p^*$ es el menor entero positivo n tal que $M^n = 1$ módulo p . Define una función `orden(M, p)` que calcule el orden de M módulo p .
2. Volvemos a nuestro problema de, dados M y MC , calcular e' tal que $M^{e'} = MC = M^e$. Un primer enfoque, fuerza bruta, consiste en elevar M a exponentes sucesivos, siempre realizando los cálculos módulo p , empezando en 1 hasta encontrar MC . Define una función de Sage `fuerza_bruta(p, M, MC)` que devuelva el exponente e' calculado en la forma indicada.
3. Hay un método mejor:
 - a) Comenzamos generando una lista, L , con los elementos M^j de \mathbb{Z}_p^* con $0 \leq j < \text{ceil}(\sqrt{p})$ (recordad que `ceil(x)` es el menor entero mayor que el decimal x).
 - b) Calculamos $m = M^{-c}$ (con $c = \text{ceil}(\sqrt{p})$) en \mathbb{Z}_p^* . Calculamos los elementos $m_i = MC \cdot m^i$ (en \mathbb{Z}_p^*), comenzando en $i = 0$ y aumentando i de uno en uno, y parando el proceso cuando encontramos un elemento m_{i_0} que ya está en la lista L , y, por tanto es de la forma M^{j_0} . De la igualdad

$$MC \cdot m^{i_0} = M^{j_0}$$

podemos despejar $MC = M^{c \cdot i_0 + j_0}$, y, por tanto, el exponente e' que hay que devolver es $c \cdot i_0 + j_0$, módulo $p-1$ debido al teorema pequeño de Fermat.

El algoritmo descrito se llama *baby step-giant step*. Define una función de SAGE `baby_giant(p, M, MC)` que lo implemente.

4. Ni `fuerza_bruta` ni `baby_giant` devuelven necesariamente el exponente e que es la clave, sino un exponente e' tal que $M^{e'} = MC = M^e$. Como $M^{e'} = M^e$ lo que podemos afirmar es que $M^{e'-e} = 1$, y, por tanto, que $e' - e$ es un múltiplo del orden de M . Esto significa que no hemos averiguado la clave e pero si su clase de restos módulo el orden de M . Con esta información es mucho más sencillo descryptar otro mensaje encriptado con la clave e mediante *fuerza bruta*, es decir, simplemente probando con elementos de la clase de restos que hemos determinado en lugar de probar con todos los enteros entre 2 y $p - 2$.
5. Queremos comprobar que las dos funciones son correctas, es decir, que para una cierto número de primos p , mensajes M y exponentes e , cuando calculamos `fuerza_bruta(p, M, M^e)`, o `baby_giant(p, M, M^e)`, siempre obtenemos un e' con la propiedad $M^{e'} = M^e = MC$. Efectúa esas comprobaciones tomando $p = \text{next_prime}(10^j)$, $j = 3, 4, 5, 6$, para cada p de los anteriores $M = \text{randint}(2, p-1)$ y e un elemento aleatorio de la lista

```
P=[j for j in xrange(2,p-1) if gcd(e,p-1)==1].
```

6. Ahora queremos comparar los tiempos de cálculo de las dos funciones. Comprueba, con los mismos datos del apartado anterior, que `baby_giant` es más eficiente que `fuerza_bruta`. Es importante, en estas comparaciones de tiempos, que las dos funciones usen el mismo conjunto de parámetros, y como algunos de los parámetros son aleatorios conviene generar primero, para cada j , los parámetros (p, M, M^e) y luego comparar tiempos usándolos.

Se recomienda, aunque no es imprescindible, hacer los apartados 3 y 4 mediante funciones de Sage, por ejemplo que dependan del parámetro j , lo que permitiría modificar fácilmente las comprobaciones.

Ejercicio 5.

El algoritmo *baby step-giant step*, del que se trata en el ejercicio anterior, es uno entre varios de los propuestos para resolver el problema que se conoce como el *cálculo del logaritmo discreto*: dados M y $MC = M^e$, calcular e' tal que $M^{e'} = MC$ en \mathbb{Z}_p^* .

En esta [página de la Wikipedia](#) se discute el problema del logaritmo discreto, para el que no se conoce solución en que el tiempo de cálculo dependa polinomialmente del tamaño de los datos, y se presentan otros algoritmos para resolverlo. Este ejercicio consiste en seleccionar alguno, quizá todos ellos, de estos algoritmos y programarlo.

Ejercicio 6.

Implementar el cifrado y descifrado del siguiente sistema de **autoclave**: la clave es una letra, por ejemplo H , y se cifra el mensaje cifrando la primera letra con César y clave H , para la segunda letra se usa César con clave la primera letra del mensaje, etc. Para descifrar basta conocer la primera clave, en este caso la H . ¿Es seguro? ¿Cómo se haría un **autovigenere**? ¿Sería mucho más seguro que el Vigenere estándar que hemos estudiado?

Ejercicio 7.

Supongamos que tenemos una clave muy secreta, un entero, que queremos que sólo se pueda conocer con la información de que disponen, entre todos, k miembros cualesquiera de un grupo de n personas. Es decir, queremos dar a cada uno de los n miembros de una comisión una subclave diferente de forma que con k subclaves cualesquiera se pueda obtener la clave total.

En la práctica, es posible implementar este método de forma que la clave da acceso a un ordenador, o a un sistema informático, pero los miembros de la comisión nunca llegan a conocerla y las subclaves están contenidas en tarjetas criptográficas de las que es muy difícil extraer la información. De esa manera ninguno de los miembros de la comisión conoce realmente ni la clave ni las subclaves.

Una solución interesante a este problema es [el método de Shamir](#), que utiliza de forma muy astuta un **polinomio interpolador**. Lee la descripción del método e implementala mediante una función que reciba el secreto y los enteros n y k , y nos devuelva las n claves parciales.

Si $k - 1$ de los miembros de la comisión deciden hacer trampa, ¿tendrían alguna ventaja para intentar un ataque de fuerza bruta por conocer $k - 1$ subclaves?

Ejercicio 8.

El PIN de las tarjetas de crédito consiste habitualmente en un entero de 4 dígitos que los bancos dicen no conocer. ¿Cómo puede funcionar?

El banco genera dos primos muy grandes, p y q , y los multiplica obteniendo $n = p \cdot q$. Entonces graba en la banda, o el chip, de la tarjeta el número p y todo el número q menos sus últimas 4 cifras, que serán el PIN.

Una vez hecho esto, olvida los primos p y q pero mantiene su producto n , de forma que cuando reconstruimos, usando el PIN y la tarjeta los enteros p y q , el banco puede multiplicarlos y comparar con el n que guarda. El banco puede no conocer el PIN porque para obtener p y q hay que factorizar n .

Define una función que genere el par de primos p y q de tal forma que Sage no pueda factorizar su producto en un tiempo de, digamos, treinta minutos.

Ejercicio 9.

Los elementos invertibles para el producto en \mathbb{Z}_n forman un grupo que habitualmente se denota por \mathbb{Z}_n^* . Su cardinal es $\phi(n)$, con ϕ la función de Euler, y sus elementos son las clases de restos con un representante primo con n . Supongamos que n

es un primo p , de forma que $\phi(p) = p - 1$, y se sabe que, en este caso, \mathbb{Z}_p^* es un grupo cíclico. Este ejercicio trata de encontrar generadores de \mathbb{Z}_p^* , mediante *fuerza bruta*, es decir, no usamos un algoritmo específico, sino que probamos distintos elementos, elegidos aleatoriamente, hasta que encontramos uno que es un generador.

1. Define una función *generador*(p) que primero compruebe si p es primo, y si lo es devuelva un generador del grupo \mathbb{Z}_p^* . Por supuesto, no sirve usar una función predefinida en Sage que ella misma calcule generadores.
2. Define una función *es_generador*(g, n) que devuelva **True** si g es un generador de \mathbb{Z}_n^* y **False** si no lo es. En este apartado, y los siguientes, n es un entero no necesariamente primo.
3. Define una función *es_ciclico*(n) que devuelva **True** si \mathbb{Z}_n^* tiene algún generador, es decir, es cíclico, y **False** si no existe generador.
4. Finalmente, ejecuta la función *es_ciclico*(n) para $2 \leq n \leq 100$, y enuncia la regla, que puedas deducir del experimento, sobre los valores de n , aparte de n primo, para los que \mathbb{Z}_n^* tiene un generador.

Ejercicio 10.

En este ejercicio construimos el sistema criptográfico de clave pública **El Gamal**, que utiliza de manera esencial el ejercicio anterior. Para simplificar el ejercicio prescindimos de la fase de codificación y nos centramos en el encriptado y desencriptado. Los mensajes codificados van a ser enteros, o más bien clases de restos, y la codificación y decodificación serían similares a las utilizadas en RSA.

1. CLAVES: La clave pública de Alice está formada por tres enteros (p, g, A) con p un primo muy grande, g un generador de \mathbb{Z}_p^* y $A := g^a \bmod p$, con el exponente $1 \leq a \leq p - 2$ que será la clave privada de Alice, que ella elige y mantiene en secreto.
2. ENCRIPTADO: Bob, que quiere encriptar un mensaje para Alice, codifica su mensaje como un entero m módulo p , elige un exponente aleatorio $1 \leq b \leq p - 2$ y calcula $B := g^b \bmod p$ y también $c := A^b \cdot m \bmod p$. El mensaje encriptado es el par (B, c) .
3. DESENCRIPTADO: Alice recibe (B, c) y define $x := p - 1 - a$, lo que le permite recuperar m como $B^x \cdot c \bmod p$.
1. Usaremos $p = \text{nth_prime}(1000)$. Elige claves para Alice. Define una función *encripta*(p, g, A, m) que devuelva (B, c) .

2. Define una función *descripta*(p, a, B, c), que devuelva el mensaje descifrado m , y comprueba que si se encripta el mensaje $m = \text{randint}(p//2, p - 2)$ se recupera el mensaje m al descriptar.
3. El exponente a no cambia, mientras Alice no cambie sus claves tiene que usar siempre a como clave privada, pero el exponente b debe ser elegido aleatoriamente cada vez que Bob quiere encriptar un mensaje para Alice. Supongamos que Bob encripta m usando como exponente b , pero el malo consigue acceder a m y a c . Posteriormente Bob encripta m' usando el mismo exponente b . Demuestra, mediante cálculos en \mathbb{Z}_p , que si el malo intercepta c' puede calcular m' sin conocer a .

Capítulo 9

Más teoría de números

Para poder utilizar la criptografía RSA con seguridad es crucial:

1. Poder comprobar de manera eficiente si un número muy grande es primo o no. Es esto lo que nos permite encontrar los dos primos muy grandes p y q cuyo producto forma parte de la clave pública del usuario.

En la práctica NO hace falta una demostración matemática completa de que los enteros p y q son primos, y basta con lo que llamamos *criterios probabilísticos de primalidad*, como el de Miller-Rabin.

2. Convencerse de que, en el estado actual de la tecnología, no es posible factorizar el entero n en un tiempo razonable. Todo entero n se puede factorizar probando posibles divisores hasta llegar a \sqrt{n} , pero para n muy grande el tiempo requerido sería tan grande que no tendría ningún sentido intentarlo.

En este capítulo, que complementa el capítulo 6, estudiamos un criterio sencillo, pero muy útil, de primalidad y un par de algoritmos de factorización de enteros. Para completar la imagen, también tratamos los mismos problemas para polinomios en una variable con coeficientes enteros o racionales.

9.1. ¿Cómo encontrar primos muy grandes? Miller-Rabin (1980)

El teorema pequeño de Fermat afirma que si n es un primo entonces $a^{n-1} \equiv 1 \pmod{n}$ para cada entero a primo con n . Si encontramos un a , primo con n , tal que el resto de dividir a^{n-1} entre n no es 1 podemos estar seguros de que n NO es primo.

Sin embargo, hay números que son compuestos pero verifican que $a^{n-1} \equiv 1 \pmod{n}$ para cada entero a primo con n , es decir, que son compuestos pero se comportan como primos para el teorema pequeño de Fermat. Estos números se llaman de *Carmichael*.

Ejercicio 1. *Escribe código, lo más eficiente posible, que, dado un entero N , devuelva una lista de los enteros de Carmichael menores que N .*

El test de Miller-Rabin está muy relacionado con el teorema pequeño de Fermat, pero es un “test probabilístico” que puede demostrar que un número compuesto lo es con una probabilidad tan grande como queramos, por ejemplo 0.999999999999. Si el test no puede probar que el número es compuesto podemos estar “casi seguros” de que es primo, y se usa en criptografía como si lo fuera. El test se basa en dos teoremas:

1. *Sea n un número primo, tal que $n - 1 = 2^s d$ con d impar, y a un entero primo con n . Entonces, o bien $a^d \equiv 1 \pmod{n}$, o bien hay un entero r en $\{0, 1, 2, \dots, s-1\}$ tal que $a^{2^r d} \equiv -1 \pmod{n}$. Si encontramos un a primo con n tal que no se cumple ninguna de las dos condiciones, podemos estar seguros de que el número n es compuesto, y decimos que el entero a es un “testigo” de la no primalidad de n .*
2. *Si $n \geq 3$ es impar y compuesto entonces el conjunto $\{1, 2, \dots, n-1\}$ contiene a lo más $(n-1)/4$ enteros que son primos con n y no son testigos de que n es compuesto.*

Supongamos que queremos “probar” que n es primo. Elegimos al azar un entero a en el conjunto $\{1, 2, \dots, n-1\}$, si resulta que a no es un testigo se ha producido un hecho que tenía probabilidad menor que $1/4$. Si repetimos t veces la elección al azar de a y nunca encontramos un testigo se ha producido un hecho de probabilidad $(1/4)^t$ que podemos hacer tan pequeña como queramos tomando t suficientemente grande, y $1 - (1/4)^t$ es la probabilidad de que si el número es compuesto Miller-Rabin lo detecte.

Cada vez que ejecutamos el test de Miller-Rabin, que depende de las elecciones al azar de los enteros a , podemos obtener resultados diferentes. Si el resultado es que el entero es compuesto, podemos estar totalmente seguros de que lo es, en cambio

si el resultado obtenido es que el entero es primo sólo tenemos una probabilidad, que puede ser todo lo próxima a 1 como queramos, de que realmente lo sea.

EN EL CAPÍTULO 11 VEREMOS LOS RUDIMENTOS DE LA TEORÍA DE LA PROBABILIDAD, de forma que si encuentras dificultad para entender las líneas básicas del argumento de Miller-Rabin, excluyendo la demostración de los dos teoremas citados, debes volver a leerlo después de haber pasado por ese capítulo.

Ejercicio 2.

1. *Escribe código para aplicar el test de Miller-Rabin a un entero n que detecte números compuestos con probabilidad p fijada. Debes, entonces, programar una función `miller-rabin(n,p)`, que dado un entero N , devuelva una lista de los enteros de Carmichael menores que N .*
2. *Escribe otra función que, dada la probabilidad p , busque enteros, dentro de un rango fijado *a priori*, que pasen el test de Miller-Rabin con probabilidad p , es decir, Miller-Rabin con esa p los declare primos, pero sean compuestos. Cuando p es muy próximo a 1, esta función devolverá probablemente una lista vacía.*
3. *Intenta aplicar el test de Miller-Rabin a un producto de dos primos muy grandes.*
4. *Usa Miller-Rabin para buscar el primo más grande que es menor que 2^{400} .*

Puedes ver una solución de estos ejercicios en la hoja de Sage [2?-TNUME-miller-rabin.sws](#).

9.1.1. AKS

En 2002 se descubrió primer test de primalidad (AKS) para números enteros n para el que el tiempo de ejecución es menor que polinomial en n . El algoritmo no es, ni mucho menos, tan complicado como cabría esperar y puedes leer sobre él en [el artículo original](#) o bien en [este otro divulgativo](#).

Los algoritmos cuya tiempo de ejecución es polinomial en el tamaño de los datos son, en principio, los algoritmos que es posible implementar de manera eficiente. Sin embargo, puede ocurrir que los coeficientes, sobre todo el coeficiente del término de mayor grado, del polinomio sean tan grandes que el algoritmo no sea útil en la práctica. Esto es lo que pasa, de momento, con el test AKS de primalidad.

9.2. ¿Cómo factorizar un entero grande?

9.2.1. Fermat

En este ejercicio vamos a estudiar un método de factorización de enteros debido a *Pierre de Fermat*. El método se basa en que si podemos escribir un entero $n > 2$ en la forma $n = x^2 - y^2$, con x e y enteros positivos, podemos con seguridad factorizar $n = (x + y)(x - y)$ que será una factorización no trivial de n si $x - y > 1$. Además, es claro que x^2 debe ser mayor que n , de forma que podemos empezar a probar posibles valores (enteros) de x empezando en $\text{ceil}(\text{sqrt}(n))$ y aumentando de uno en uno. Si para un x dado encontramos que $x^2 - n$ es un cuadrado habremos encontrado una solución de la ecuación $n = x^2 - y^2$ y, por tanto, una factorización, que puede ser trivial.

Supongamos que $n = c \cdot d$ es una factorización no trivial con $c \geq d$ y que n es impar. Entonces c y d son impares y

$$n = c \cdot d = \left(\frac{c+d}{2}\right)^2 - \left(\frac{c-d}{2}\right)^2,$$

de forma que la factorización puede ser obtenida por el método de Fermat. En consecuencia, si el método produce la factorización trivial, $n = n \cdot 1$, entonces el número n es con seguridad primo.

Ejercicio 3.

1. DEFINE UNA FUNCIÓN **bruta(n)** que encuentre una factorización no trivial de n , si es que existe, y si no existe devuelva la trivial. El método debe consistir en probar la divisibilidad entre los enteros menores o iguales a \sqrt{n} . La función debe devolver los dos factores de n , c y d , y la diferencia $n - c \cdot d$.

ES CLARO que esta función será eficiente si el número n tiene factores primos pequeños, pero muy ineficiente si todos sus factores son muy grandes. Por ejemplo, si el número es de la forma p^2 con p un primo muy grande, este algoritmo debe llegar hasta la prueba final para encontrar la factorización $p^2 = p \cdot p$.

2. DEFINE UNA FUNCIÓN **fermat(n)** que implemente el método de factorización de Fermat y devuelva los dos factores de n , c y d , y la diferencia $n - c \cdot d$.

TAMBIÉN DEBE SER CLARO que esta función encuentra pronto una solución si hay una factorización $n = c \cdot d$, $c \geq d$ con c no demasiado distante de \sqrt{n} . VEMOS ENTONCES que los dos métodos son complementarios: donde uno es menos eficiente el otro lo es más.

3. Parece una buena idea, dado que los métodos son complementarios, mezclarlos: DEFINE UNA FUNCIÓN `fermat_bruta(n,B)` tal que

- a) Aplique el método de Fermat desde `ceil(sqrt(n))` hasta `B` (`B` debe ser mayor que `ceil(sqrt(n))`). Si encuentra una factorización la debe devolver.
- b) Si Fermat no ha funcionado, use el método del apartado 1, probando con enteros del intervalo $[2, \text{ceil}(B - \sqrt{B^2 - n})]$. ¿Por qué es suficiente probar con estos enteros?

Supongamos que queremos factorizar enteros que son el producto de dos primos. ESTUDIA, en función de los valores de `B`, los casos

- a) `n=nth_prime(396741)*nth_prime(236723)=18954671729831` en el que los dos primos difieren en 2451674.
- b) `n=nth_prime(3967741)*nth_prime(2367)=1415121961361` en el que los dos primos difieren en 67266400.

Para poder ver las diferencias conviene imprimir algún texto que nos avise de que el método de Fermat no ha encontrado solución y la función está aplicando el segundo método.

Puedes ver una solución de estos ejercicios en la hoja de Sage [2?-TNUM-fermat.sws](#).

9.2.2. Pollard $p - 1$ (1974)

El algoritmo $p - 1$ de Pollard sirve para factorizar enteros n impares que tienen un factor primo p tal que $p - 1$, que es compuesto, tiene factores primos no demasiado grandes. Necesita una cota *a priori*, B , del tamaño de los primos que aparecen en la factorización de $p - 1$. Es decir, si n es compuesto y tiene un factor primo p tal que todos los factores primos de $p - 1$ son menores que B el algoritmo puede encontrar un factor no trivial de n .

1. Se comienza eligiendo un entero M tal que $p - 1$ sea con seguridad un divisor de M . Como no conocemos la factorización en primos de $p - 1$ debemos tomar como M el producto de todos los primos menores que B con exponentes mayores que los exponentes con los que aparecen en la factorización de $p - 1$. Por ejemplo, si $p - 1 = 2^{k_1} \cdot 3^{k_2} \dots p_i^{k_i} \dots$ y como $p - 1 < n$, podemos afirmar con seguridad que $k_2 \cdot \log(2) < \log(n)$ y, en general $k_i < \log(n)/\log(p_i) = \log_{p_i}(n)$. Estas cotas, que son bastante más grandes de lo necesario, sólo dependen de n y los primos p_i y se verifica que $p - 1$ divide a M .

2. Gracias al teorema pequeño de Fermat, podemos afirmar que $a^M \equiv 1 \pmod{p}$ para todo a que sea primo con p . Elegimos entonces un a primo con n , y por tanto primo con p , y debe verificarse que p también divide a $a^M - 1$ y, por tanto, p debe dividir al máximo común divisor de $a^M - 1$ y n .
3. El máximo común divisor de $a^M - 1$ y n es un divisor de n , que es lo que buscábamos, y nos sirve si es diferente de n . La ventaja enorme es que, como debemos saber, se puede calcular módulo n .
4. Si el algoritmo no encuentra un factor n , podemos probar con diferentes valores de a , por ejemplo haciendo un cierto número de elecciones aleatorias para a , y si todavía no encontramos un factor podemos incrementar B .
5. El problema es que si repetimos con diferentes valores de a o incrementamos la cota B el tiempo de ejecución del programa aumenta, y puede ocurrir que tarde tanto como lo que tardaría un programa que probara los posibles divisores menores que la raíz cuadrada de n (el método “bestia”). A pesar de sus limitaciones, el método es bastante eficiente y ha dado lugar a toda una serie de refinamientos que son los actualmente en uso para factorizar enteros grandes.
6. Por supuesto, antes de intentar factorizar n debemos estar convencidos de que es compuesto, por ejemplo aplicándole Miller-Rabin.

Ejercicio 4.

1. Implementa el algoritmo de Pollard y comprueba su funcionamiento en un número razonable de casos.
2. El algoritmo $p - 1$ funciona bien cuando n tiene un factor primo p tal que $p - 1$ sólo tiene primos menores que un B y todos los demás factores de n son grandes respecto al correspondiente M . Trata de comprobar experimentalmente esta afirmación.

Puedes ver una solución de estos ejercicios en la hoja de Sage [2?-TNUME-pollard.sws](#).

9.3. Irreducibilidad y factorización de polinomios

El anillo de polinomios $k[x]$ en una variable con coeficientes en un cuerpo k tiene propiedades algebraicas muy similares a las del anillo de los enteros. Aparte de que existen una suma y un producto verificando las propiedades que definen *un anillo*, existe también en los dos una división con resto, o división euclidiana.

Nos interesan en este momento los polinomios con coeficientes números racionales, que podemos transformar en polinomios con coeficientes enteros sin más que multiplicar por el mínimo común múltiplo de todos los denominadores que aparecen en los coeficientes.

Los polinomios con coeficientes enteros no tienen, en general, división con resto, pero la mayor simplicidad de los coeficientes tiene algunas ventajas al estudiar su irreducibilidad o factorización.

9.3.1. Polinomios con coeficientes racionales

Supondremos las siguientes definiciones y resultados, que deberían ser conocidos gracias a la asignatura *Conjuntos y Números*:

1. Un anillo D se dice que es un *dominio* si $a \cdot b = 0$, $a, b \in D$ implica $a = 0$ ó $b = 0$.
2. Un elemento $a \in D$ se dice que es *invertible* si existe otro $b \in D$ tal que $a \cdot b = 1$. El b mencionado, el *inverso* de a , se suele denotar mediante a^{-1} .
3. Se dice que un elemento $a \in D$ *divide* a otro $b \in D$, con la notación $a \mid b$, si existe un un tercero $c \in d$ tal que $a \cdot c = b$.
4. Un elemento, no invertible, a en un dominio D se dice *primo* si $a \mid b \cdot c$ implica $a \mid b$ ó $a \mid c$.
5. Un elemento, no invertible, $a \in D$ se dice *irreducible* si $a = b \cdot c$ implica que b o c es invertible.
6. El anillo de polinomios en una variable, con coeficientes en un cuerpo k cualquiera, es un dominio $k[x]$ en el que existe división con resto y el grado del polinomio resto es menor que el grado del divisor.
7. En $k[x]$ los elementos invertibles son los polinomios constantes distintos del cero, y los elementos primos son los mismos que los irreducibles.

Todo polinomio en $k[x]$ es el producto de un número finito de polinomios primos, que, además, son únicos salvo el orden de los factores y el producto por elementos invertibles.

9.3.2. Polinomios con coeficientes enteros

En este caso también hay que recordar algunas definiciones y resultados:

1. Los polinomios con coeficientes enteros, $\mathbb{Z}[x]$ forman un dominio. Las definiciones de elemento invertible, divisibilidad, elemento primo y elemento irreducible se pueden aplicar sin cambio. Los únicos elementos invertibles son ± 1 .
2. Se llama *contenido*, $c(p(x))$, de un polinomio $p(x)$ con coeficientes en \mathbb{Z} al MCD de los coeficientes de $p(x)$. Un polinomio $p(x)$ tal que $c(p(x)) = 1$ se dice *primitivo*.
3. Sea $p(x) \in \mathbb{Z}[x]$ un polinomio primitivo de grado > 0 . Entonces $p(x)$ es irreducible en $\mathbb{Z}[x]$ si y sólo si lo es en $\mathbb{Q}[x]$. Además, los polinomios primitivos e irreducibles son los mismos que los primitivos y primos.
4. Todo polinomio con coeficientes enteros es el producto de un entero, su contenido, y un número finito de polinomios primitivos y primos (irreducibles). Esta descomposición es única salvo el orden de los factores.

Estos resultados nos garantizan que podemos estudiar la irreducibilidad y factorización de polinomios con coeficientes racionales usando el polinomio con coeficientes enteros que se obtiene multiplicando el polinomio por el MCM de los denominadores de los coeficientes.

9.3.3. Polinomios en Sage

1. Para definir el anillo de polinomios en una variable sobre los racionales usamos

```
A.<x>=QQ[]
```

en el que la variable se llama x y el anillo A .

Si queremos el anillo con coeficientes enteros usamos ZZ en lugar de QQ.

2. Definimos dos, o más, polinomios en la forma

```
F,G=x^4+1,x-7
```

y operamos con ellos usando los símbolos habituales para la suma y el producto:

```

expand(F+G)
H = expand(F^2*G^2)
HH = factor(H)

```

La segunda línea expande el producto, aplicando la propiedad distributiva, y la tercera factoriza. El resultado de una factorización es una lista de pares ordenados, cada par está formado por un factor y su multiplicidad.

3. El método **P.quo_rem**(Q), con P y Q polinomios en una variable, devuelve el cociente y el resto de la división con resto de P entre Q .
4. Dada una lista de pares de números racionales $((1, 2), (2, 7), (-1, 3))$ tal que las primeras coordenadas son todas diferentes, la instrucción

```
A.lagrange_polynomial((1,2),(2,7),(-1,3))
```

```
11/6*x^2 - 1/2*x + 2/3
```

devuelve el polinomio de menor grado tal que los puntos de la lista pertenecen a su gráfica. El grado del polinomio devuelto es igual al número de pares que deben estar en la gráfica menos 1. Se llama a este polinomio el *interpolador de Lagrange*, y su cálculo, planteado como la resolución de un sistema lineal de ecuaciones, ya fue tratado en la sección ****.

9.3.4. Factorización de Kronecker

Comenzamos describiendo el algoritmo:

1. Supongamos dado un polinomio $p(x)$ con coeficientes enteros que podemos suponer primitivo. Si se verifica $q(x) \mid p(x)$ para algún polinomio $q(x)$, necesariamente se cumplirá también $q(n) \mid p(n)$ para todo entero n . Esta es la base del método de Kronecker.
2. Supongamos que el grado de $q(x)$ es m . Sean x_0, x_1, \dots, x_m enteros distintos, que podemos suponer para simplificar los cálculos que son los enteros distintos y menores en valor absoluto que podemos elegir $(0, \pm 1, \pm 2, \dots)$.

Llamemos X_i al conjunto de todos los divisores, positivos y negativos, de $p(x_i)$ y sea X el producto cartesiano de los X_i . Lo importante aquí es que X es un conjunto finito.

3. Para cada elemento $\mathbf{b} := (b_0, b_1, \dots, b_m)$ de X podemos determinar un polinomio interpolador que toma en los x_i el valor b_i y es un candidato a divisor $q_{\mathbf{b}}(x)$ de $p(x)$. Dividimos $p(x)$ entre $q_{\mathbf{b}}(x)$ para averiguar si efectivamente es un divisor o no.

Si resulta ser un divisor repetimos el procedimiento usando el cociente en lugar de $p(x)$.

4. Hay que repetir todo el proceso anterior para cada posible grado $1 \leq m \leq n$, pero en todo caso únicamente hay que efectuar un número finito de pruebas.
5. Al final del proceso habremos encontrado explícitamente una factorización de $p(x)$ en $\mathbb{Z}[x]$ o habremos demostrado que $p(x)$ es irreducible.

El algoritmo generaliza el que utilizamos para calcular las raíces racionales de un polinomio con coeficientes racionales. De hecho, si sólo probamos posibles factores de grado 1 en el algoritmo de Kronecker obtenemos el que sirve para encontrar raíces racionales.

Aunque demuestra que el problema de factorizar un polinomio con coeficientes racionales puede ser resuelto en un número finito de pasos, no es de gran utilidad práctica porque en cuanto el grado de $p(x)$ es alto el número de polinomios $q(x)$ que hay que probar si dividen a $p(x)$ es enormemente alto.

Sistemas como Sage utilizan algoritmos mucho más sofisticados, que, en general, se basan en estudiar las factorizaciones de los polinomios que se obtienen reduciendo los coeficientes módulo una potencia de un primo.

Ejercicio 5. *Implementa el algoritmo de Kronecker y estudia sus límites. Compara los resultados con el método de factorización de polinomios que se incluye en Sage.*

Puedes ver una solución de este ejercicio en la hoja de Sage [2?-TNUME-kronecker.sws](#).

9.3.5. Criterio de irreducibilidad de Brillhart

Si un polinomio con coeficientes enteros $P(x)$ toma para un cierto entero m un valor primo ($P(m) = p$ con p primo) parece poco probable que el polinomio pueda ser reducible: si lo fuera, $P(x) = P_1(x) \cdot P_2(x)$, tendría que verificarse $P_1(m) = \pm 1$ o bien $P_1(m) = \pm p$. El criterio de Brillhart nos suministra una condición suficiente para que, teniendo un valor primo, el polinomio sea irreducible.

Dado un polinomio con coeficientes enteros

$$P(x) := a_0 + a_1x + a_2x^2 + \cdots + a_nx^n,$$

definimos

$$M := \max_{0 \leq i \leq n-1} \left\{ \frac{|a_i|}{|a_n|} \right\}.$$

Tenemos entonces

(Criterio de Brillhart) Supongamos que existe un $m \in \mathbb{Z}$, $m \geq M + 2$, tal que $P(m)$ es primo. Entonces el polinomio $P(x)$ es irreducible en $\mathbb{Z}[x]$.

Puedes ver una demostración del criterio de Brillhart en su [artículo original](#). No es difícil, y es muy recomendable.

Ejercicio 6.

1. Define una función de Sage que reciba como argumentos un polinomio con coeficientes enteros $P(x)$ y un entero N_0 y devuelva `True` si existe un entero m , menor que N_0 y mayor que $M + 1$, tal que $P(m)$ es primo.
2. Queremos estudiar hasta qué punto el criterio de Brillhart nos permite decidir si un polinomio con coeficientes enteros es irreducible. Para eso queremos estimar, por ejemplo, la fracción $B(N, N_0)/I(N)$ con $I(N)$ el número de polinomios enteros de grado 5, con coeficientes entre $-N$ y N , primitivos e irreducibles, y $B(N, N_0)$ el número de los que podemos probar que son irreducibles usando Brillhart con $m \leq N_0$. Es interesante tratar de ver si esa fracción se estabiliza cuando hacemos crecer N y N_0 .

Este problema puede ser atacado usando el tipo de paralelismo vergonzante que incluye Sage (ver la sección ****).

Puedes ver una solución de estos ejercicios en la hoja de Sage [2?-TNUME-brillhart.sws](#).

Capítulo 10

Matemática discreta

La Matemática discreta es, simplemente, la parte de las Matemáticas que trata de objetos finitos, o en ocasiones numerables. Decimos que esos objetos son discretos por oposición a objetos *continuos* como los números reales.

Así, por ejemplo, podemos decir que el estudio de los grupos finitos está en la intersección del Álgebra y la Matemática discreta, o que el de los planos finitos lo está en la de la Geometría y la Matemática discreta.

Es claro que usando ordenadores únicamente podemos llegar a estudiar *objetos discretos*, finitos o numerables, y únicamente podemos aproximarnos a los objetos continuos mediante su *discretización*, viendo, por ejemplo, un número real como un decimal con un número dado de cifras decimales.

En este capítulo comenzamos con un resumen de la combinatoria básica, que trata, esencialmente, del siguiente problema: suponemos dada una familia de conjuntos finitos $X_{n,k}$, que depende de parámetros enteros, en este caso n y k .

Se trata de encontrar fórmulas, que debemos determinar,

$$\#(X_{n,k}) = F(n, k)$$

que nos den el número de elementos de $X_{n,k}$ como una función F de los parámetros. Es decir, el problema central de la combinatoria consiste en contar el número de elementos de conjuntos en función de parámetros.

Como no siempre se ha visto combinatoria en el Bachillerato, se puede consultar un pequeño resumen en [este enlace](#). Cubre la parte más básica, permutaciones, variaciones y combinaciones, junto con algunos aspectos más avanzados que serán útiles

al resolver los ejercicios de este capítulo.

En segundo lugar, se incluye una breve introducción a la teoría de grafos, que son quizá las estructuras finitas con mayor número de aplicaciones a muy diversos problemas.

10.1. Permutaciones, variaciones y combinaciones

1. PERMUTACIONES: Dado un conjunto X con n elementos, el conjunto de todas las listas de longitud n que podemos formar usando todos los elementos del conjunto es el conjunto de *permutaciones* de X .

El número de permutaciones de un conjunto de n elementos es (¿por qué?)

$$n! := n \cdot (n-1) \cdot (n-2) \dots 3 \cdot 2 \cdot 1.$$

En Sage podemos generar el conjunto de permutaciones de $X := \{1, 2, \dots, n\}$ mediante `p=Permutations(n)` y formar una lista L con todas las permutaciones usando `L=p.list()`.

2. VARIACIONES: Dado un conjunto X con n elementos, el conjunto de todas las listas de longitud k que podemos formar usando k elementos distintos del conjunto es el conjunto de *variaciones* de X usando k elementos.

El número de variaciones de X usando k elementos es $n!/(n-k)!$ (¿por qué?).

En Sage podemos generar el conjunto de variaciones de $X := \{1, 2, \dots, n\}$ usando k elementos mediante `v=Permutations(n, k)` y formar una lista L con todas las variaciones usando `L=v.list()`.

3. COMBINACIONES: Dado un conjunto X con n elementos, el conjunto de todos los subconjuntos de longitud k que podemos formar con elementos de X es el conjunto de *combinaciones* de X usando k elementos.

El número de combinaciones de X usando k elementos es $\binom{n}{k} := \frac{n!}{k!(n-k)!}$ (¿por qué?).

En Sage podemos generar el conjunto de combinaciones de $X := \{1, 2, \dots, n\}$ usando k elementos mediante `c=Subsets(n, k)` y formar una lista L con todos los subconjuntos usando `L=c.list()`. Los elementos de esta lista son conjuntos no listas, y si fuera necesario convertirlos podríamos usar `[list(s) for s in L]`.

Si especificamos un segundo argumento, `c=Subsets(n)` se genera el conjunto de todos los subconjuntos de X , es decir, el conjunto de partes de X con número de elementos 2^n .

10.2. Funciones entre conjuntos finitos

Una aplicación biyectiva, $f : A \longrightarrow B$, entre dos conjuntos, es una aplicación al tiempo *inyectiva* y *sobreyectiva*. Una aplicación es *inyectiva* si para cualesquiera dos elementos, $a_1, a_2 \in A$, tales que $f(a_1) = f(a_2)$ necesariamente $a_1 = a_2$. Una aplicación es *sobreyectiva* si para cualquier elemento $b \in B$ existe un elemento $a \in A$ tal que $f(a) = b$.

Recordemos que:

1. Dadas funciones $f : A \longrightarrow B$ y $g : B \longrightarrow C$ su *composición* $g \circ f : A \longrightarrow C$ es la función $g \circ f(a) := g(f(a))$.

Si dos aplicaciones biyectivas se pueden componer, la composición es, de nuevo, una biyección.

2. En todo conjunto A hay una biyección $1_A : A \longrightarrow A$ definida mediante $1_A(a) := a$ para todo $a \in A$.

Toda biyección $f : A \longrightarrow B$ tiene una *inversa* $g : B \longrightarrow A$, verificando $g \circ f = 1_A$ y $f \circ g = 1_B$, que es también una biyección.

3. Una función $f : A \longrightarrow B$ es inyectiva si y sólo si existe una $g : B \longrightarrow A$ tal que $g \circ f = 1_A$.

4. Una función $f : A \longrightarrow B$ es suprayectiva si y sólo si existe una $g : B \longrightarrow A$ tal que $f \circ g = 1_B$.

En lo que sigue vamos a utilizar ciertas estructuras de datos, listas y diccionarios, para representar en el ordenador las funciones y poder trabajar con ellas.

Para empezar puedes consultar la hoja [101-MDISC-biyecciones.sws](#).

En estos ejercicios se entiende que *construir* un conjunto de funciones es definir una función de Sage que devuelva, por ejemplo, una lista con elementos todas las funciones que se quiere construir. Construir estos conjuntos, que pueden ser enormes, permite atacar ciertos problemas mediante *fuerza bruta*. Como sabemos, este método de resolver problemas está limitado por la cantidad de memoria RAM disponible y el tiempo que estemos dispuestos a esperar una solución.

Por otra parte, hemos visto que podemos usar como clave en un cierto sistema criptográfico una biyección del alfabeto en sí mismo, o, usando el **truco para oscurecer las frecuencias**, una biyección de un conjunto mucho mayor en sí mismo.

Ejercicio 1. Construye el conjunto $F(X, X)$ de todas las funciones de un conjunto finito $X := \{0, 1, 2, \dots, n-1\}$ en sí mismo. Cada función debe venir dada por un diccionario.

Ejercicio 2. Construye el subconjunto de las biyecciones de X en X y una función de Sage que para cada biyección nos devuelva la biyección inversa.

Ejercicio 3. Dado dos conjuntos $X_n := \{0, 1, 2, \dots, n-1\}$ e $Y_m := \{0, 1, 2, \dots, m-1\}$ con $m < n$ construye el conjunto $F(X_n, Y_m)$ de todas las funciones de X_n en Y_m .

Ejercicio 4. Construye el subconjunto de todas las funciones suprayectivas de X_n en Y_m . ¿Cuántas hay?

Llamemos $Sup(n, m)$ al número de funciones suprayectivas de X_n en Y_m . Una fórmula que determine $Sup(n, m)$ como una función de n y m es bastante complicada, pero es más fácil obtener una fórmula recursiva que dé $Sup(n+1, m)$ en función de $Sup(n, m)$ y $Sup(n, m-1)$. DEDUCE esa fórmula y COMPRUEBALA con ayuda del programa realizado en este apartado, es decir, programa la fórmula recursiva y comprueba que produce los mismos resultados que contar el número de funciones suprayectivas.

Ejercicio 5. Dada una función suprayectiva $f : X \rightarrow Y$ construye el conjunto de funciones de $g : Y \rightarrow X$ tales que $f \circ g = 1_Y$.

Ejercicio 6. Sea \mathbb{Z}_m el conjunto de las clases de restos módulo un entero m , que representaremos como la lista `srange(m)`. Consideramos la función $\Phi_k(n) : \mathbb{Z}_m \rightarrow \mathbb{Z}_m$ definida mediante $\Phi_k(n) := (k * n) \% m$. Queremos caracterizar los pares (m, k) tales que Φ_k es biyectiva. Si no conoces la respuesta debes experimentar con distintos pares (m, k) hasta entender lo que pasa, y si la conoces el ejercicio consiste en producir código para comprobarla.

REPITE el ejercicio, pero con la función $\Psi_k(n) := (k^n) \% m$.

Puedes encontrar soluciones en la hoja `102-MDISC-funciones-sol.sws`.

10.3. Acciones de grupos

Dados un grupo G y un conjunto X , decimos que G actúa en X si hemos definido una función $\mu : G \times X \rightarrow X$ tal que $\mu(1, x) = x$, para todo $x \in X$, y $\mu(g_1, \mu(g_2, x)) = \mu(g_1 \cdot g_2, x)$.

Habitualmente se denota la función μ mediante un producto, $\mu(g, x) =: g \cdot x$ que hay que distinguir cuidadosamente del producto en el grupo G . La primera propiedad en la definición de una acción de un grupo en un conjunto puede enunciarse diciendo que el elemento neutro en el grupo, $1 \in G$ es *neutro para la acción*, y la segunda es una *propiedad asociativa*.

El conjunto de órbitas X/G para una acción de G en X es el conjunto cociente X para la relación de equivalencia

$$x_1 \sim x_2 \text{ si y sólo si existe } g \in G \text{ tal que } x_2 = g \cdot x_1.$$

Cada clase de equivalencia, $[x]$, es la órbita $o(x)$ de uno cualquiera de sus elementos x , es decir,

$$[x] = o(x) = \{g \cdot x \mid g \in G\}.$$

Estas nociones, *acción de un grupo en un conjunto y conjunto de órbitas*, son importantes en geometría con X un conjunto de figuras geométricas, por ejemplo triángulos o cónicas planas, y G un grupo, por ejemplo el grupo de transformaciones afines del plano, que utilizamos para clasificar las figuras.

Así, todos los triángulos son equivalentes respecto al grupo afín y tenemos una única órbita, pero no todas las cónicas son equivalentes respecto al mismo grupo: podemos transformar una circunferencia en una elipse pero no una circunferencia en una parábola.

HEMOS USADO la palabra *órbita* para referirnos al conjunto de los iterados de un elemento $x \in X$ mediante una función $f : X \rightarrow X$:

$$o(x) := \{x, f(x), f^2(x) := f(f(x)), f^3(x) := f(f(f(x))), \dots\},$$

y la relación con una órbita de una acción de un grupo debe ser clara: *si f es biyectiva* forma junto con todos sus iterados, con exponente positivo o negativo, un grupo

$$G := \{\dots, f^{-2}, f^{-1}, f^0 = \text{identidad}, f, f^2, f^3, \dots\},$$

y la órbita de x para este grupo es la unión de la órbita de x para f con la órbita de x para la función inversa f^{-1} .

10.4. Grafos

Los grafos son objetos geométricos 1-dimensionales:

Un grafo $G = (V, E)$ dirigido con conjunto de vértices V finito es un subconjunto $E \subset V \times V$.

Entonces, un grafo puede verse como un conjunto de *vértices* V junto con un conjunto de pares ordenados de vértices, que representamos como flechas que unen el primer vértice del par con el segundo, con la flecha apuntando al segundo, y llamamos *ejes* del grafo. Podemos entonces representar el grafo mediante un dibujo, por ejemplo en el plano, y verlo como un objeto geométrico 1-dimensional.

Un grafo con conjunto de vértices V finito es un subconjunto $E \subset V \times V/\mathbb{Z}_2$, con $V \times V/\mathbb{Z}_2$ el conjunto órbitas para la acción de $\mathbb{Z}_2 := \{1, \sigma\}$ en $V \times V$ definida por $\sigma(v_1, v_2) := (v_2, v_1)$.

En el caso de un grafo, no dirigidos, representamos los ejes como segmentos sin la flecha, y cuando queramos indicar que un grafo está dirigido lo indicaremos explícitamente.

ALGUNAS DEFINICIONES:

1. Un eje que une un vértice consigo mismo se dice que es un *lazo*.
2. Un *camino* en el grafo desde el vértice v al v' es una sucesión de ejes e_1, e_2, \dots, e_n tales que e_1 contiene a v , e_n a v' y el eje e_i tiene un vértice común con e_{i+1} para $1 \leq i < n$.

Si el grafo está dirigido necesitamos además que las flechas apunten todas de v a v' .

3. Un *ciclo* en el grafo es un camino cerrado, es decir, tal que v coincide con v' .
4. Un *árbol* es un grafo sin ciclos.
5. Un grafo es *conexo* si para cada par de vértices existe un camino que los une. Un grafo no conexo es una *unión disjunta* de grafos, que llamamos las *componentes*, de forma que no hay ejes entre una componente y otra distinta.
6. Un grafo se dice que es *planar* si puede ser representado en el plano sin que se corten sus ejes.
7. Un grafo es *euleriano* si existe un ciclo que pasa una única vez por cada eje.
8. Un grafo es *hamiltoniano* si existe un ciclo que contiene una única vez cada vértice.
9. Denotamos los vértices de un grafo como $V := \{v_1, v_2, \dots, v_n\}$. La *matriz de adyacencia* del grafo es la matriz M $n \times n$ con entradas ceros y unos definida mediante

$$M_{ij} = \begin{cases} 1 & \text{si hay un eje que una } v_i \text{ con } v_j \\ 0 & \text{si no lo hay} \end{cases}$$

La matriz de adyacencia contiene toda la información sobre el grafo, y mediante el producto de matrices permite extraer nueva información. La matriz de un grafo no dirigido es simétrica.

10. Dos grafos son *isomorfos* si existe una biyección entre sus conjuntos de vértices de forma que dos ejes están unidos mediante un eje en el primer grafo si y sólo si sus imágenes están unidas mediante un eje en el segundo.

Es decir, dos grafos son isomorfos si son el mismo salvo las etiquetas de los vértices.

11. Llamamos *grado de un vértice* al número de ejes que contienen al vértice.
12. Un grafo se dice *regular* si todos los vértices tienen el mismo grado.
13. El *grafo completo* con n vértices es el que tiene todos los ejes posibles, es decir, su conjunto de ejes es $V \times V$.

Ejercicio 7. Sea G un grafo, dirigido o no, con matriz de adyacencia M .

1. Demuestra que la entrada $(M^n)_{ij}$ es un entero que cuenta el número de caminos con n ejes, dirigidos si el grafo es dirigido, desde el vértice v_i al vértice v_j .
2. Demuestra que el número de triángulos, ciclo con tres ejes, contenidos en el grafo es un sexto de la traza de M^3 .

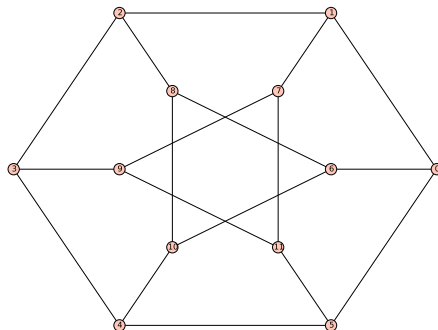
10.4.1. Grafos en Sage

1. CONSTRUCCIÓN DE GRAFOS:

- a) Sage conoce una gran cantidad de grafos que tienen nombre. Se puede ver la lista escribiendo en una celda `graphs` y pulsando la tecla del tabulador. Por ejemplo,

```
G = graphs.DurerGraph()
show(G)
```

produce este grafo:



- b) La instrucción `1 = graphs.RandomNP(n,p)` produce un grafo aleatoriamente construido con n vértices y probabilidad p de incluir cada uno de los posibles ejes. Cuando p es baja obtendremos casi siempre un grafo no conexo mientras que cuando es alta será casi seguramente conexo.

- c) La instrucción

```
G2 = graphs.RandomGNM(n,m)
```

produce un grafo aleatoriamente elegido con n vértices y m ejes. Elegimos el grafo, de entre todos los grafos con n vértices y m ejes, de manera equiprobable.

- d) Podemos generar un grafo usando su matriz de adyacencia:

```
M1 = matrix(ZZ,7,[randint(0,1) for muda in srange(49)])
M = (M1+M1.transpose()) / 2
G3 = Graph(M)
show(G3)
```

Como se trata de un grafo no dirigido, su matriz debe ser simétrica y simetrizamos la matriz $M1$, que ha sido generada aleatoriamente, en la segunda línea.

- e) También podemos dar explícitamente la lista de ejes:

```
G4=Graph(7) #7          vertices
```

```
ejes=[(1,3),(2,5)] #Una lista de pares (duplas) de vertices
G4.add_edges(ejes)
show(G4)
```

2. MÉTODOS PARA GRAFOS:

- a) Como en general con Sage, podemos ver todos los métodos que se aplican a los grafos creando un grafo, por ejemplo con nombre G , y pulsando el tabulador en una celda en la que hemos escrito G .. La cantidad de métodos existentes es enorme, y en este curso sólo llegaremos a usar unos pocos.
- b) Disponemos de métodos, con respuesta booleana, para comprobar si un grafo verifica una cierta propiedad. Son los métodos del tipo $G.is_...$, como por ejemplo
 - 1) $G.is_connected()$.
 - 2) $G.is_eulerian()$.
 - 3) $G.is_hamiltonian()$.
 - 4) $G.is_planar()$.
 - 5) $G.is_tree()$.
 - 6) $G.is_regular()$.
 - 7) $G1.is_isomorphic(G2)$.

Si el grafo G resulta ser euleriano, podríamos ver un ciclo euleriano mediante **show**($G.eulerian_circuit()$), y en el caso hamiltoniano mediante **show**($G2.hamiltonian_cycle()$).

- c) Podemos obtener la matriz de adyacencia de un grafo G mediante $M=G.adjacency()$, de forma que, después de ejecutar esa instrucción, podemos operar con la matriz M .

Ejemplo 1. Encontrar el mayor subconjunto de enteros entre 1 y 100 con la propiedad de que no hay dos de ellos cuya diferencia sea, en valor absoluto, un cuadrado.

Se puede resolver mediante este código

```
n=100
G=Graph(n)
```

```
cuadrados=[i*i for i in xrange(sqrt(n))]  
ejes=[(i,j) for (i,j) in CartesianProduct(xrange(n),xrange(n)) if (i!=j and  
abs(i-j) in cuadrados)]  
G.add_edges(ejes)  
L = G.independent_set();L
```

teniendo en cuenta que `G.independent_set()` devuelve un conjunto maximal de vértices, i.e. no podemos añadir ningún vértice sin que deje de tener la propiedad que enunciamos a continuación, tal que no hay ejes entre ningún par de vértices del subconjunto.

Capítulo 11

Probabilidad

La ciencia intenta conseguir, siguiendo el modelo de la mecánica de Newton, capacidad predictiva absoluta, es decir, pretende que dadas las condiciones iniciales, la situación en el instante $t = 0$, sea posible calcular la evolución futura, e incluso en algunos casos la evolución previa del sistema objeto de estudio.

Esta pretensión ha resultado ser en gran medida ilusoria:

1. No es posible medir con precisión absoluta las condiciones iniciales, y pequenísimas variaciones en esas medidas pueden afectar drásticamente al resultado.
2. Muy pocos de los problemas matemáticos a que da lugar la ciencia son exactamente resolubles. Las ecuaciones, diferenciales o en derivadas parciales, que aparecen son demasiado complicadas y, aunque podemos estudiar sus soluciones, en muy pocos casos sabemos resolverlas exactamente.

Se puede usar *Cálculo numérico* para resolver de manera aproximada las ecuaciones, pero se producen errores de redondeo que no es fácil controlar.

3. La ciencia a escala atómica, y subatómica, es intrínsecamente probabilista. En Mecánica cuántica no podemos predecir el resultado de muchos experimentos y la teoría únicamente nos da probabilidades de los resultados posibles.

Esta situación no se debe a nuestra incapacidad para calcular mejor, como en el caso de la mecánica clásica, sino que está en la naturaleza de las cosas.

Entonces, nos interesamos por el resultado de *experimentos aleatorios*, es decir, experimentos cuyo resultado no consideramos predecible con certeza. Un ejemplo puede ser el experimento consistente en el lanzamiento de una moneda: en principio, las leyes de la mecánica se podrían aplicar y deberían permitir calcular, de manera exacta, la trayectoria de la moneda y el resultado, cara o cruz, del experimento, pero en la práctica tal cálculo es imposible, y debemos conformarnos con mucho menos.

En la teoría de la probabilidad *nos conformamos con observar las regularidades que aparecen al repetir el experimento un gran número de veces*. Cuando lanzamos una moneda, un gran número de veces, observamos que casi siempre, aproximadamente, la mitad de los resultados son caras y la mitad cruces, y cuando una moneda no se ajusta a este comportamiento diríamos que está trucada.

En este capítulo calculamos probabilidades mediante simulación usando generadores de números aleatorios. Hay muchos problemas que podemos simular, y sorprende que con estos métodos se consigan tan buenas aproximaciones al resultado exacto calculado de acuerdo a la teoría de la probabilidad. Comprobaremos esto en el caso de la probabilidad de obtener un número de caras dado al lanzar repetidas veces una moneda.

El objetivo del capítulo es introducir un par de ideas básicas de la teoría y, sobre todo, intentar mejorar nuestra intuición de los fenómenos probabilísticos.

11.1. Probabilidad

Llamemos X al conjunto de “resultados atómicos”, es decir, resultados que no podemos descomponer como unión de otros resultados, de un experimento aleatorio. Suponemos primero que el conjunto X es finito, aunque todo esto se puede generalizar, y sus elementos son los *sucesos elementales (atómicos)*. A un subconjunto $A \subset X$ le llamamos *suceso* sin más.

Llamemos n al número de elementos de X . Cuando realizamos el experimento en la realidad, un número N de veces, obtenemos, para cada uno de los sucesos elementales x_i , una *frecuencia* f_i que, por definición, es el número de veces m_i que obtenemos el resultado $x_i \in X$ dividido por el número N de repeticiones del experimento.

Es claro que

$$\sum_{i=1}^n f_i = \sum_{i=1}^n \frac{m_i}{N} = \frac{\sum_{i=1}^n m_i}{N} = \frac{N}{N} = 1.$$

En experimentos aleatorios en los que podemos controlar de manera precisa las condiciones principales que determinan el resultado se observa que, al crecer N , las frecuencias tienden a estabilizarse.

La base de la teoría de la probabilidad es entonces un espacio X de sucesos elementales, que todavía suponemos finito, y una función $p : X \rightarrow \mathbb{R}$ que asigna “probabilidades” p_i a los sucesos elementales x_i y que verifica

1. Para todo i

$$0 \leq p_i = p(x_i) \leq 1.$$

2.

$$\sum_{i=1}^n p_i = 1.$$

Supuesto que ya tenemos X y p , definimos entonces la probabilidad de un suceso $A \subset X$ como

$$p(A) := \sum_{x_i \in A} p_i.$$

En muchos casos, PERO NO SIEMPRE, los sucesos elementales tienen todos la misma probabilidad (i.e. son equiprobables), que entonces vale $1/n$, y para la probabilidad $p(A)$ de un suceso $A \subset X$ obtenemos, usando la notación $\#(Y)$ para el número de elementos del conjunto Y ,

$$p(A) = \frac{\#(A)}{\#(X)},$$

que es la famosa fórmula $p(A) = \text{casos favorables} / \text{casos posibles}$.

Podemos pensar, aunque ésta no es la única interpretación posible, las probabilidades p_i como el límite, cuando N tiende a infinito, de las frecuencias f_i .

En este curso *simularemos experimentos aleatorios* un número grande de veces N y obtendremos frecuencias que, cuando N crece, calculan, experimentalmente, las probabilidades que nos interesan.

Si dos sucesos, A y B , son disjuntos, $A \cap B = \emptyset$, entonces la probabilidad de que se dé uno o el otro debe ser la suma de las probabilidades, ya que, en esas condiciones

$$\text{casos favorables a } A \cup B = \text{casos favorables a } A + \text{casos favorables a } B,$$

y los casos posibles son siempre los mismos.

Por otra parte, la probabilidad de que se verifiquen simultáneamente A y B , $p(A \cap B)$, es, en muchos casos igual al producto $p(A) \cdot p(B)$, pero no siempre. Se dice que *los sucesos A y B son independientes si y sólo si*

$$p(A \cap B) = p(A) \cdot p(B).$$

En muchos casos es natural suponer que ciertos sucesos son independientes, de acuerdo a la definición anterior, ya que no vemos que el hecho de que uno se verifique influya en el otro. Así, por ejemplo, debemos suponer que si lanzamos una moneda varias veces el resultado de cada lanzamiento es independiente de los otros. Sin embargo, supongamos un dado con tres caras pintadas de rojo, las del 1, 2 y 3, y las restantes tres de verde. La probabilidad de obtener verde y 1 es cero, pero la de 1 es $1/6$ y la de verde $1/2$, luego no son sucesos independientes.

11.1.1. Variables aleatorias

Una **variable aleatoria** es una función $\mathbb{X} : X \rightarrow \mathbb{R}$, con X un espacio de sucesos elementales tal que algunos de sus subconjuntos $A \subset X$ tienen asignada una probabilidad $p(A)$. No entramos en detalles técnicos acerca de las propiedades que deben cumplir X y \mathbb{X} , que se verán más adelante en los cursos de Probabilidad.

Por ejemplo, si lanzamos dos dados de distinto color, el espacio X de sucesos elementales tiene 36 elementos y se define una variable aleatoria considerando, para cada suceso elemental, la suma de los puntos obtenidos en ese suceso. La variable aleatoria toma valores enteros entre 2 y 12, y nos interesa, por ejemplo para apostar sobre los diversos resultados, calcular la probabilidad de sucesos como $\mathbb{X} \geq 7$, es decir la probabilidad de obtener 7 o más puntos lanzando dos dados.

Decimos que una variable aleatoria $\mathbb{X} : X \rightarrow [0, 1] \subset \mathbb{R}$ es **uniforme** si, para cada par $0 \leq a \leq b \leq 1$, la probabilidad del suceso $a \leq \mathbb{X} \leq b$ es exactamente $b - a$.

La simulación de variables aleatorias uniformes es clave en todo lo que sigue.

11.1.2. Simulación

En esta subsección simulamos, usando el generador de números aleatorios de Sage, lanzamientos de una moneda correcta y de una moneda trucada. La base de estas simulaciones, y de otras muchas que veremos, es el hecho de que el generador **random()** simula una variable aleatoria uniforme en el intervalo $[0, 1]$.

1. Comenzamos simulando el lanzamiento de una moneda equiprobable un cierto número de veces:

```
[randint(0,1) for n in xrange(10)]
```

```
[0, 1, 0, 0, 1, 0, 0, 1, 1, 1]
```

```
(([randint(0,1) for n in xrange(10)]).count(1)
```

```
5
```

```
(([randint(0,1) for n in xrange(10^5)]).count(1)
```

```
50202
```

```
time sum([( [randint(0,1) for n in xrange(10^5)]).count(1) for int in
xrange(10)])
```

```
498868
```

```
Time: CPU 12.51 s, Wall: 12.52 s
```

La instrucción `randint(a,b)` produce un entero en el intervalo cerrado $[a, b]$ que es “aleatorio”, en el sentido de que todos los enteros del intervalo son igualmente probables.

En la última línea, repetimos 10 veces el experimento de lanzar una moneda 100000 veces, y observamos que el número de cruces es, en promedio, 49886.8 que es bastante próximo a 50000.

2. Queremos ahora simular una MONEDA TRUCADA, de forma que, por ejemplo, la probabilidad de cara sea sólo $1/3$. Para eso, usamos una VARIABLE ALEATORIA UNIFORME en el intervalo $[0, 1]$ y observamos cuántos valores de la variable caen en el intervalo $[0, 1/3]$.

```
random()
```

```
0.46257246456300749
```

```
def moneda_trucada(p):
    x = random()
    if x <= p:
        return 0
    else:
        return 1
```

```
moneda_trucada(1/3)
```

```
1
```

```
([moneda_trucada(1/3) for n in xrange(10^5)]).count(1)
```

```
66684
```

La instrucción `random()` produce un decimal, de la precisión que indiquemos, perteneciente al intervalo $[0, 1]$. Por supuesto, el resultado es siempre un número racional pero los reales no racionales no existen para la máquina.

3. Técnicamente, los números que producen las dos instrucciones son PSEUDOALEATORIOS ya que están producidos mediante un procedimiento determinista, iterando una función matemática que tiene que tener un período muy alto. Además la distribución de valores tiene que ser muy próxima a uniforme, es decir, `randint` debe producir enteros en el intervalo con (casi) la misma frecuencia todos y `random` debe producir decimales en el intervalo $[0, 1]$ con frecuencia de pertenencia a cada subintervalo (casi) igual a la longitud del subintervalo.

Ejercicio 1.

1. Calcula, experimentalmente, la probabilidad de obtener a lo más 100 caras en 1000 lanzamientos de una moneda para la que la probabilidad de cara es $\frac{1}{10} \cdot k$, $k = 1, 2, \dots, 9$. Por supuesto, este ejercicio tiene también una respuesta “teórica”, pero, de momento, no nos interesa.
2. ¿Cómo producir números aleatorios con distribución uniforme en un intervalo $[a, b]$ cualquiera?

3. LA RUINA DEL JUGADOR: Un jugador acude a un casino con 100 euros y acepta el siguiente juego: se lanza una moneda y si sale cara gana un euro del casino y si sale cruz paga un euro al casino. Suponemos que el casino dispone de una cantidad ilimitada de dinero.
- a) Comprueba que, en las condiciones anteriores, la ruina del jugador es segura.
 - b) Supongamos ahora que el jugador decide retirarse cuando su saldo llega a 200 euros o bien a 50 euros. Calcula la probabilidad de que se retire habiendo ganado.

11.2. Binomial

El experimento aleatorio más sencillo es el lanzamiento de una moneda, y el siguiente más sencillo es lanzar una moneda un número N de veces. Acabamos de ver cómo simular, usando números (pseudo)aleatorios, el lanzamiento de una moneda o incluso el de una moneda trucada, y en el ejercicio 1 comenzamos a utilizar esa capacidad de simular para responder a algunas cuestiones (quizá) interesantes.

En esta sección discutimos la justificación teórica del resultado obtenido mediante simulación al resolver el primer apartado.

El ejercicio pide que se calcule la probabilidad de obtener a lo más 100 caras al lanzar 1000 veces una moneda trucada que tiene probabilidad 0.1 de cara.

1. Comenzamos discutiendo la probabilidad de obtener exactamente 100 cara. La sucesión de mil lanzamientos se puede representar como una cadena de longitud 1000 de ceros y unos (cero para cara y uno para cruz). En cada lanzamiento de la moneda la probabilidad de obtener cara es $1/10$ y lanzamientos sucesivos son *independientes*, el resultado de los anteriores no influye en los que siguen. En estas condiciones, las probabilidades *se multiplican*, por ejemplo la probabilidad de obtener dos caras en las primeras dos tiradas es $1/100$.

La probabilidad de obtener 100 caras en las primeras 100 tiradas es $(1/10)^{100}$, y la probabilidad de que el resto de las 1000 tiradas sean cruces es $((1/10)^{100})((9/10)^{900})$. Para cada uno de las posibles cadenas de ceros y unos que contienen 100 ceros y 900 unos la probabilidad de aparición de esa cadena particular es la misma, e igual a $((1/10)^{100})((9/10)^{900})$.

2. ¿Cuántas cadenas hay que contengan 100 ceros y 900 unos? Basta elegir las 100 posiciones en las que colocamos ceros, ya que el resto van a ser unos necesariamente. Una ordenación tiene todos los ceros al comienzo, y hay $1000!$ reordenaciones

de esta primera, pero muchas son iguales, por ejemplo, si intercambiamos el primer cero con el segundo queda la misma cadena. ¿Cuántas reordenaciones de la cadena que tiene los cien ceros al principio sigue teniendo los cien ceros al principio? Todas las que permuten los cien primeros ceros entre sí o los 900 unos entre sí. El número de esas reordenaciones es $100! \times 900!$, de forma que el número de cadenas con cien ceros y novecientos unos es

$$\binom{1000}{100} := \frac{1000!}{100! \times 900!}.$$

Entonces la probabilidad de obtener exactamente 100 caras es

$$\binom{1000}{100} ((1/10)^{100}) ((9/10)^{900}),$$

que es aproximadamente igual a 0.042.

3. Entonces, la solución a nuestro problema original es

$$\sum_{i=0}^{i=100} \binom{1000}{i} (0.1)^i (0.9)^{1000-i},$$

que resulta ser, aproximadamente, 0.5266.

En la hoja [111-PROBA-ejercicio-1.sws](#), que contiene soluciones para el ejercicio 1, puedes ver que la simulación consigue, en un tiempo razonable, un resultado con 3 cifras decimales correctas.

11.2.1. Caso general

Supongamos que repetimos N veces el lanzamiento de una moneda que tiene probabilidad p de cara, por tanto, $1 - p$ de cruz. Como en la discusión anterior asignamos 0 al resultado “cara” y 1 al resultado cruz.

Este experimento aleatorio produce una variable aleatoria $X = \text{“número de caras obtenidas al lanzar la moneda } N \text{ veces”}$, definida sobre el conjunto (finito) de cadenas binarias de longitud N y con valores enteros entre 0 y N .

Una cadena binaria particular, que contenga i ceros, es, como hemos visto, un suceso atómico con probabilidad $p^i(1-p)^{N-i}$. Nos interesa calcular la probabilidad de obtener exactamente i caras, y hemos visto que, como hay $\binom{N}{i}$ cadenas distintas con exactamente i ceros, la probabilidad de obtener i ceros es

$$\binom{N}{i} p^i (1-p)^{N-i}.$$

Esta variable aleatoria X se dice que *es binomial*.

11.3. Generadores de números (pseudo-)aleatorios

Supongamos que queremos programar una función como `random()` que es uniforme en el intervalo $[0, 1]$. Los números reales en el intervalo los veremos, en el ordenador, como decimales, por ejemplo, con diez cifras. Si los multiplicamos por 10^{10} los podemos ver como enteros en el intervalo $[0, 10^{10}]$, o como clases de restos módulo $n := 10^{10} + 1$.

Los generadores (pseudo-)aleatorios más simples funcionan iterando una función adecuada f de \mathbb{Z}_N , con N muy grande, en sí mismo. El primer número aleatorio que se genera, digamos x_0 , se llama la semilla, y salvo que indiquemos explícitamente lo contrario, el ordenador la genera cada vez usando, por ejemplo, el reloj interno de la máquina.

Los siguientes se obtienen iterando la función f , es decir, mediante $x_n := f^n(x_0)$. Es claro que este proceso no tiene de “aleatorio” sino la elección de la semilla, el resto es completamente “determinista”.

1. Cuando un generador de este tipo vuelve a un valor ya visitado, por ejemplo en x_m , es decir $x_m = x_n$ con $n < m$, se produce un período: a partir de m los números generados son los mismos que se generaron a partir de n , y ya no sirven para nada útil. Los buenos generadores son los que tienen períodos de longitud enormemente grande y que además visitan casi todos los elementos de \mathbb{Z}_N antes de caer en un período.
2. La instrucción `random()` genera números en el intervalo $[0, 1]$ de manera (aproximadamente) uniforme, es decir, la frecuencia con que el resultado, en M repeticiones, pertenece al subintervalo $[a, b]$ debe ser muy próxima a $b - a$. Una manera fácil de comenzar a estudiar la calidad de los generadores es, simplemente, comprobar si esta propiedad de uniformidad se mantiene cuando M crece. En general, se utilizan métodos *estadísticos*, los mismos que sirven para estudiar muestras de datos obtenidas del mundo real, para estudiar la uniformidad de los números suministrados por distintos generadores (pseudo-)aleatorios.

Los generadores descritos no pueden tener períodos mayores que el entero n (¿por qué?), pero una variante que utiliza funciones de varias variables, por ejemplo k , y determina el valor x_{n+1} usando los k últimos números puede alcanzar períodos mucho más altos. Los generadores modernos son de este tipo. Por ejemplo, el generador que utiliza Python, y también Sage, llamado *Mersenne Twister*, produce enteros aleatorios de 32 bits con un período de longitud $2^{19937} - 1$, es decir N es 2^{32} pero el período del generador es mucho mayor.

Los generadores de números (pseudo-)aleatorios tienen un uso importante en la práctica de la criptografía ya que sirven para elegir claves aleatoriamente en un espacio de posibles claves que debe ser enorme para que el sistema sea seguro. Parece ser que el método que utiliza (¿utilizaba?) la NSA (*National Security Agency* de Estados Unidos) para descifrar masivamente mensajes se basa en que forzaron la utilización en el *software* criptográfico de *generadores de números aleatorios trucados*. Es claro, por ejemplo, que si el usuario cree que está usando claves de 2048 bits, que se consideran suficientemente seguras, pero de hecho el generador elige las claves en un espacio mucho menor, por ejemplo 512 bits, la seguridad es totalmente ficticia.

La hoja de [111-PROBA-generador-vn.sws](#) contiene un estudio de uno de los generadores primitivos, inventado por von-Neumann.

11.3.1. Generación eficiente de números aleatorios

El generador de números (pseudo-)aleatorios que utilizan, por defecto, Python y Sage es, como acabamos de ver, el llamado **Mersenne Twister**, y se considera actualmente como uno de los mejores. Cuando debemos generar millones de números aleatorios conviene hacerlo directamente en C ya que, como sabemos, Python se interpreta no se compila y es esencialmente lento.

Mediante Cython podemos utilizar las siguientes funciones que llaman al mismo generador pero calculando en C:

```
%%cython
cdef extern from 'gsl/gsl_rng.h':
    ctypedef struct gsl_rng_type:
        pass
    ctypedef struct gsl_rng:
        pass
    gsl_rng_type *gsl_rng_mt19937
    gsl_rng *gsl_rng_alloc(gsl_rng_type * T)
```



```

cdef gsl_rng *r = gsl_rng_alloc(gsl_rng_mt19937)

cdef extern from 'gsl/gsl_randist.h':
    long int uniform 'gsl_rng_uniform_int'(gsl_rng * r, unsigned long int n)

def main():
    cdef int n
    n = uniform(r, 1000000)
    return n

```

Ejercicio 2.

1. Programa y estudia este [generador](#), en particular estudiando sus períodos y autocorrelaciones. Se puede usar como modelo la hoja sobre el generador de vonNeumann.
2. En la sección 4 del [artículo](#) se describe el generador conocido como *Blum-Blum-Shub*. Impleméntalo en Sage y estudia alguna de las afirmaciones que se hacen en el artículo acerca de él.
3. Explora la [lista](#) de generadores de números pseudo-aleatorios, y programa en Sage los que te interesen.

11.4. Elementos aleatorios

En Sage existe el método `A.random_element()` que devuelve un elemento aleatoriamente elegido, es decir, de manera equiprobable, dentro del objeto finito A . El método no está implementado para cualquier tipo de objeto, y cuando no existe y lo necesitamos debemos programarlo.

Comenzamos con los casos más sencillos: `A.random_element()` no existe si A es una lista o un conjunto.

```

def elemento_aleatorio(A):
    '''A es una lista o conjunto'''
    B = list(A)
    n = len(B)-1
    return B[randint(0,n)]

```

11.4.1. Grafos aleatorios

¿Qué es un grafo aleatorio?

1. Fijado el número n de vértices, y una vez que ponemos etiquetas a los vértices, podríamos decir que un *grafo aleatorio* es uno elegido al azar, de forma equiprobable, entre todos los grafos etiquetados con n vértices. Como fijado el número de vértices hay un número finito, aunque muy grande, de grafos con ese número de vértices, la definición tiene sentido.

Con mayor propiedad, diríamos que se trata de un *grafo aleatoriamente elegido*. Uno de los problemas con esta noción es que quizá nos interese más considerar como equiprobables las clases de isomorfismo de grafos, es decir, considerar dos grafos como iguales si difieren únicamente en el etiquetado.

También es posible considerar grafos aleatoriamente elegidos fijando el número de vértices y el de ejes. Esta noción de *grafo aleatorio* es la que corresponde a *elemento aleatorio del conjunto de grafos con n vértices y e ejes*.

2. Otra posible noción de *grafo aleatorio* sería el que construimos generando cada posible eje con una probabilidad p fijada de antemano e independientemente de los ejes que ya tenga el grafo, es decir, para cada par de vértices lanzamos una moneda con probabilidad p de cara y si sale cara ponemos en el grafo el eje que une ese par de vértices y si sale cruz no.

Diríamos que este es un *grafo construido aleatoriamente*.

Las dos nociones son distintas ya que, por ejemplo, en la segunda y si p es muy baja el grafo resultante tendrá muy pocos ejes.

En Sage es fácil obtener un grafo, con n vértices, construido aleatoriamente con probabilidad p : usamos la instrucción `G = graphs.RandomGNP(n,p)` y podemos ver el grafo resultante con `show(G)`.

Ejercicio 3. En este ejercicio debemos estudiar la probabilidad, cuando n tiende a infinito, de que un grafo con n vértices construido aleatoriamente con probabilidad $p = \frac{2\log(n)}{n}$ sea conexo (el logaritmo es el neperiano).

11.5. Simulación: método de Monte Carlo

Los métodos de *Monte Carlo* se caracterizan por el uso masivo de números aleatorios para calcular el comportamiento de ciertos sistemas complejos. En muchos casos hablamos de *simulación de Monte Carlo*, es decir, nuestro programa simula el comportamiento de un sistema complejo, por ejemplo, de un número grande de partículas en movimiento.

En esta sección veremos varios ejemplos.

11.5.1. Cálculo aproximado de π

Como π es el área de un círculo de radio unidad, la parte del círculo de centro el origen y radio unidad que está en el primer cuadrante tiene área $\pi/4$ y está contenida en el cuadrado $[0, 1] \times [0, 1]$ de área unidad.

Ejercicio 4.

1. Si elegimos un número N , muy grande, de números aleatorios ejecutando N veces `random()`, debemos esperar, debido a que un buen generador debe producir resultados uniformes, que haya aproximadamente $N/10$ de ellos en cada uno de los subintervalos $[i/10, (i+1)/10]$, $i = 0, \dots, 9$. De la misma forma, si hemos obtenido n_i números aleatorios en el subintervalo $[i/100, (i+1)/100]$ ($i = 0, \dots, 99$), debe ser n_i aproximadamente igual a $N/100$ para todos los i .

COMPRUEBA que cuando N crece

$$\sum (n_i - \frac{N}{100})^2$$

decrece.

2. Digamos que un punto aleatorio del cuadrado $[0, 1] \times [0, 1]$ es uno con coordenadas `(random(), random())`. COMPRUEBA, de forma similar a lo hecho en el apartado anterior, la uniformidad de la distribución cuando generamos N puntos aleatorios en el cuadrado.
3. Como la distribución de N puntos aleatorios en el cuadrado es uniforme, podemos CALCULAR una aproximación a $\pi/4$ determinando cuántos puntos, de los N , caen dentro de la circunferencia. La fracción, respecto a N , de los que caen dentro debe ser aproximadamente igual al área, $\pi/4$, del sector.

Puedes ver una solución, del apartado 3, en la hoja [114-PROBA-pi-paralelo.sws](#).

11.5.2. Cálculo de áreas

Podemos, en principio, calcular aproximadamente el área de una figura acotada, es decir, contenida en un rectángulo, usando la misma técnica que para la circunferencia. La única dificultad reside en el proceso de decidir si el punto está dentro o fuera de la figura.

Puedes ver una serie de ejemplos en la hoja [115-PROBA-areas.sws](#).

11.5.3. Cálculo de integrales

Calcular integrales de manera exacta no es siempre fácil, y en muchos casos debemos conformarnos con un resultado aproximado. Hay varios métodos para calcular aproximadamente el valor de integrales.

1. Se puede obtener una buena aproximación, tomando N muy grande, al valor de una integral considerando la suma

$$\frac{b-a}{N} \sum_{i=1}^N f(x_i),$$

donde hemos dividido el intervalo $[a, b]$ en N subintervalos de longitud $\frac{b-a}{N}$ y tomamos x_i como el extremo superior del subintervalo i -ésimo.

2. MONTE CARLO:

```
def integral(f,a,b,N):
    return ((b-a)/N).n()*sum([f(x=a+(b-a)*random()) for muda in srange(N)])
```

Cuando N es muy grande, por ejemplo podemos imponer que $(b-a)/N$ sea menor que 10^{-5} , este procedimiento calcula bastante bien el valor de la integral. ¿Por qué?

- a) Estamos subdividiendo el intervalo $[a, b]$ en N subintervalos iguales, cada uno de longitud $(b-a)/N$, de forma que podemos sacar $(b-a)/N$ factor común en la fórmula para $S_\Delta(f, a, b)$.
- b) Producimos N números aleatorios en el intervalo $[a, b]$ y debemos esperar que, más o menos, caiga uno en cada subintervalo. Se pueden producir pequeños errores en el cálculo debido a que en algunos subintervalos caigan varios de los números aleatorios y en otros ninguno.
- c) El valor de f en uno de esos números aleatorios es el valor en un punto cualquiera del subintervalo.
- d) En consecuencia, el valor que devuelve **integral**(f,a,b,**N**) es una suma de Riemann con subintervalos de longitud uniforme $(b-a)/N$, y sabemos que el valor de la integral es el límite, cuando la longitud de los subintervalos tiende a cero, de las sumas de Riemann. Entonces, el valor devuelto es una buena aproximación a la integral cuando N es muy grande.

Usando este método se calculan aproximadamente integrales, que casi siempre tienen dos o tres cifras decimales correctas. Aumentando N mucho se puede mejorar la precisión del cálculo.

El método de Monte Carlo para calcular integrales es especialmente útil en el caso de integrales de funciones de varias variables, en el que puede ser en ocasiones el mejor de los métodos disponibles.

Sin embargo, también es importante, en ciertas aplicaciones, obtener buenas aproximaciones al valor de una integral tomando muy pocos subintervalos (N pequeño). Este problema forma parte del *Cálculo Numérico*.

11.5.4. Cálculo de probabilidades

Comenzamos con un ejemplo curioso. Es el llamado “truco de Monty Hall”: se trata de un concurso de TV¹ en el que el concursante debe elegir entre tres puertas sabiendo que detrás de una de ellas hay un gran premio y detrás de las otras esencialmente nada. Una vez que el concursante ha elegido una puerta, el presentador, Monty Hall en el concurso en EE.UU., hace abrir una de las puertas que no tienen premio y ofrece al concursante cambiar de puerta o reafirmarse en la primera elección. ¿Es mejor cambiar o no cambiar?

El truco consiste en que el concursante puede pensar que, una vez que ya sólo quedan dos puertas, hay una probabilidad $1/2$ de que el premio esté detrás de cada una de las puertas, de forma que para él es indiferente mantener la elección o cambiar. El concursante puede ver el ofrecimiento de cambiar de puerta como *presión por parte del presentador para cambiar*, perdiendo el premio que el presentador sabe que está detrás de la puerta elegida, y tiende a aferrarse a su elección original, sin embargo la probabilidad de ganar cambiando es bastante mayor que $1/2$.

¹En TVE se importó la idea dentro del programa *Un, dos, tres... responde otra vez*, que comenzó en 1972 y se mantuvo durante 10 temporadas.

```
def jugada_mh(eleccion):
    me_cambio = 1  # (1)
    puertas = [0]*3
    puertas[randint(0,2)] = 1  # (2)
    quedan = puertas  # (3)
    del quedan[eleccion]  # (4)
    la_otra = (1 in quedan)  # (5)
    if me_cambio == 1:
        return la_otra
    else:
        return puertas[eleccion]  # (6)
```

1. Cambiar a 0 para ver la probabilidad de ganar el premio manteniendo la elección original.
2. El premio esta en **randint**(0,2).
3. Copia de la situación de premios detras de las puertas.
4. Borramos la elección original de la lista **quedan**, que ahora tiene longitud 2.
5. Booleano: vale 1 si el premio no estaba detrás de la puerta elegida inicialmente.
6. Mantengo mi elección original.

En el código no es necesario implementar explícitamente el hecho de que una de las puertas que no tiene premio ha sido abierta, ya que si 1 está en la lista **quedan** cambiando se va a obtener premio y si 1 no está en **quedan** cambiando no se va a obtener premio.

```
sum([jugada_mh(1) for int in xrange(100000)])
```

66683

Vemos que las probabilidades de obtener el premio cambiando de puerta o no cambiando no son iguales, es decir las dos $1/2$, sino que la probabilidad de obtener el premio cambiando de puerta es del orden de $66683/100000$, es decir más o menos $2/3$.

Puedes leer algo más sobre el problema en este [artículo de la Wikipedia](#).

Ejercicio 5.

1. En este hipotético juego de casino apostamos un Euro a la posibilidad de que en 1000 lanzamientos de una moneda correcta se obtengan 10 caras seguidas. Supongamos que en caso de ganar el casino nos da como premio X Euros y queremos averiguar el valor de $X > 1$ que hace que el juego sea justo².

DEBEMOS CALCULAR LA PROBABILIDAD p de que en 1000 lanzamientos de una moneda correcta se obtengan 10 caras seguidas, ya que el valor justo de X es el que hace que $p \cdot X = 1$, de forma que el jugador, si juega mucho, termine aproximadamente con la misma riqueza que al comienzo. Si se cumple esta condición, el casino tampoco obtendría, a largo plazo, ninguna ganancia.

2. N personas asisten a una reunión todos con un sombrero parecido, y al terminar se llevan un sombrero elegido al azar. Estima la probabilidad de que ninguno se lleve su propio sombrero.
3. Estima la probabilidad de que en una reunión en la que hay N personas haya 2 al menos que cumplen años el mismo día.

Puedes ver una solución de estos ejercicios en la hoja [116-PROBA-probabilidades.sws](#).

11.5.5. Un sistema físico

En este ejercicio simulamos un sistema físico muy simple y nada realista, pero de gran interés. Supongamos una lista L de longitud n que en el momento inicial tiene en todas sus entradas el valor 5. Cada una de esas entradas representa una “partícula” que en el momento inicial $t = 0$ tiene una “energía” igual a 5 unidades. El sistema L evoluciona en el tiempo, es decir para $t = 0, 1, 2, 3, \dots$ obtenemos listas $L_0 = L, L_1, L_2, L_3, \dots$, de la siguiente manera:

1. Dada la lista L_t elegimos una entrada al azar, es decir con igual probabilidad para cada una de las n entradas. Supongamos que hemos obtenido la entrada i .

²En realidad los juegos de los casinos son siempre injustos para el jugador, que si juega mucho acabará arruinado, porque si no los casinos no serían rentables y desaparecerían.

2. A continuación elegimos al azar otra entrada de L_t y obtenemos la entrada j .
3. Si $L_t[i]$ es mayor que cero, definimos $L_{t+1}[i] := L_t[i] - 1$ y $L_{t+1}[j] := L_t[j] + 1$, y dejamos las demás entradas igual, y si $L_t[i] = 0$ dejamos $L_{t+1} = L_t$. Es decir, la partícula i ha “interaccionado”, en el instante t , con la j y le ha transferido una unidad de energía, pero todo el tiempo la energía total del sistema es $5n$ y, por tanto, la energía media es siempre 5.

Ejercicio 6.

1. Programa una función de dos argumentos enteros n la longitud de L y N el valor máximo de t , y que devuelva la lista L_N , que representa los valores de la energía de las partículas después del paso de N “segundos”.
2. Define una lista, por ejemplo $M = [1, 2, 3]$, define $T = \text{finance.TimeSeries}(M)$, que convierte la lista en una serie temporal (el primer elemento es el correspondiente a $t = 0$, el segundo a $t = 1$, etc.) y estudia la información (poca) que se obtiene con la instrucción `T.plot_histogram()`. El gráfico que se obtiene es el “histograma” correspondiente a la serie temporal T .
3. Utiliza la información obtenida en el apartado anterior para producir, mediante un bucle **for** adecuado, una serie de histogramas correspondientes a $n = 1000$ y $N = 100, 1000, 10000, 100000, 1000000$. ¿Qué observas en los histogramas acerca de la evolución temporal del sistema de partículas?

Puedes ver una solución de este ejercicio en la hoja [11?-PROBA-particulas.sws](#).

11.5.6. Cadenas de Markov

Comenzamos con un ejercicio sencillo:

Ejercicio 7. En cada día la Bolsa, por ejemplo de Nueva York, puede subir, bajar o mantenerse. Supongamos que las probabilidades de transición, de un día al siguiente, entre cada uno de estos tres estados son:

1. Si un día el mercado subió tenemos probabilidades 0.3 de que suba al siguiente, 0.5 de que baje y 0.2 de que se mantenga.
2. Si un día el mercado bajó tenemos probabilidades 0.4 de que suba al siguiente, 0.3 de que baje y 0.3 de que se mantenga.
3. Si un día el mercado se mantuvo tenemos probabilidades 0.4 de que suba al siguiente, 0.4 de que baje y 0.2 de que se mantenga.

Obsérvese que la suma de las tres probabilidades, en cada uno de los tres casos, es 1 ya que necesariamente el mercado debe subir, mantenerse o bajar.

Se trata de analizar el comportamiento a largo plazo de este mercado. Más concretamente, si observamos el mercado durante un período largo de tiempo, ¿cuál es la probabilidad de que lo encontremos en cada uno de los tres estados (subiendo, bajando, manteniéndose)?

Puedes ver una solución de este ejercicio en la hoja [11?-PROBA-mercado.sws](#), y una lista de ejercicios propuestos en [11?-PROBA-markov.sws](#), con soluciones en [11?-PROBA-markov-sol.sws](#).

11.5.7. Paseos aleatorios

El problema de la ruina del jugador, que ya hemos estudiado, es un ejemplo de *paseo aleatorio*. Otro ejemplo similar, que es el que da nombre a los paseos aleatorios, consiste en imaginar un paseante que comienza, en tiempo $t = 0$, en el origen de la recta real, y en cada instante de tiempo se desplaza una unidad entera hacia la derecha o la izquierda de manera equiprobable. En todo momento el paseante está en un punto de coordenada entera.

Esta situación tan simple da lugar a muchos problemas interesantes:

Ejercicio 8.

1. ¿Cuál es, en promedio, la distancia al origen del paseante en el instante N ?
2. ¿Cuál es la probabilidad de que el paseante vuelva, en algún momento, al origen?

11.5.7.1. Paseos 2-dimensionales

En los paseos bidimensionales el paseante comienza, en $t = 0$, en $(0, 0)$ y se desplaza entre puntos de coordenadas enteras. En cada instante debe decidir entre cuatro opciones de desplazamiento: *Norte, Sur, Este, Oeste*, y decide entre ellas de manera equiprobable.

Ejercicio 9.

1. ¿Cuál es, en promedio, la distancia al origen del paseante en el instante N ?
2. ¿Cuál es la probabilidad de que el paseante vuelva, en algún momento, al origen?

3. Supongamos dos paseantes que comienzan su paseo en el instante $t = 0$ en el origen. ¿Cuál es la probabilidad de que el segundo paseante visite, en algún momento de su paseo, un lugar en el que ya ha estado el primero?
4. ¿Cuál es la probabilidad de que dos paseantes que comienzan su paseo en el instante $t = 0$ en el origen vuelvan a encontrarse (i.e. lleguen en un cierto instante $t = N$ a ocupar la misma posición en el plano)?

11.5.8. Urnas de Polya

Utilizamos una variable *tiempo* discreta que toma valores enteros $t = 0, 1, 2, 3, \dots$. La variable t sería, por ejemplo, el tiempo transcurrido medido en minutos.

Se trata de simular la siguiente situación:

En el instante $t = 0$ tenemos una urna U_0 que tiene una bola blanca y una negra. En el instante $t = n$ extraemos una bola de la urna U_{n-1} , la urna tal como estaba en el instante $t = n - 1$, y la devolvemos a la urna junto con otra del mismo color y ese es el estado U_n de la urna. En cada instante $t \in \mathbb{N}$ tenemos una probabilidad $P(t)$ de extraer una bola blanca de la urna U_t .

Si describimos la urna en el momento t mediante una lista $U_t := [b(t), n(t)]$ ($b(t)$ el número de bolas blancas y $n(t)$ el de negras en el instante t) es claro que, como la extracción se hace al azar, es decir, todas las bolas tienen la misma probabilidad de ser extraídas igual a $1/(b(t) + n(t))$, la probabilidad $P(t) = b(t)/(b(t) + n(t))$.

La función $P(t)$ depende de extracciones realizadas al azar, y no tiene sentido “determinarla”: cada vez que se realice el proceso de generar las urnas U_t se obtienen resultados bastante diferentes para $P(t)$. Sin embargo pueden aparecer “regularidades a largo plazo” ($t \rightarrow \infty$) que queremos estudiar.

Ejercicio 10.

1. Definir una función de SAGE que dada la urna $U(t)$ devuelva la urna U_{t+1} .
2. Definir una función de SAGE, dependiendo de un parámetro entero N , que devuelva una lista

$$[P(0), P(1), P(2), \dots, P(N)]$$

que simule el comportamiento de las urnas de Polya.

3. Representar gráficamente la lista obtenida en el apartado anterior. ¿Qué observas cuando se ejecuta un cierto número de veces la representación gráfica con listas diferentes.
4. Para N suficientemente grande estudia, generando n (también suficientemente grande) urnas, el aspecto de los valores $P(N)$ que se obtienen para cada una de las urnas. En particular puedes usar las funciones de SAGE
 - $T = \text{finance.TimeSeries}(L)$ necesaria para poder aplicar a T las funciones que siguen. Esta función declara que T es una “serie temporal” que es otro nombre para una función, como $P(t)$, de un tiempo discreto.
 - $T.\text{histogram}(\text{bins}=10)$ agrupa los valores de la serie temporal en 10 intervalos y cuenta las frecuencias en cada intervalo. Devuelve el par formado por la lista ordenada de las frecuencias y la lista ordenada de los intervalos.
 - $t.\text{plot}\backslash_ \text{histogram}(\text{bins}=10, \text{normalize}=\text{True})$ crea el gráfico correspondiente a la función anterior.

Puedes ver una solución de estos ejercicios en la hoja [11?-PROBA-urnas-polya.sws](#).

11.5.9. Probabilidades geométricas

Supongamos que tenemos un triángulo equilátero inscrito en la circunferencia unidad. Elegimos una cuerda “al azar” y queremos CALCULAR LA PROBABILIDAD de que la cuerda sea más larga que el lado del triángulo. Hay varios métodos para elegir la cuerda “al azar” y cada uno de ellos da un resultado diferente:

1. Elegimos los extremos de la cuerda, dos puntos elegidos al azar sobre la circunferencia, y obtenemos la cuerda uniéndolos. Para elegir los puntos podemos usar que la circunferencia se parametriza mediante $(\cos(t), \sin(t))$ con $t \in [0, 2\pi)$.
2. Elegimos un punto P al azar dentro de la circunferencia y a continuación decidimos que P es el punto medio de la cuerda. La cuerda queda totalmente determinada por esta decisión, y su longitud es mayor que la del lado del triángulo si y sólo si P está dentro del círculo de radio $1/2$ (Explica estas dos afirmaciones).
3. Elegimos un radio “al azar”, eligiendo su extremo sobre la circunferencia unidad, y decidimos que el punto medio de la cuerda es ahora un punto elegido “al azar” sobre el radio.

Ejercicio 11.

1. Simula cada uno de los tres métodos (usando `random()`) para obtener estimaciones de la probabilidad buscada. Como se indica más arriba, se obtienen valores totalmente diferentes.
2. No es difícil obtener, sobre todo *a posteriori*, argumentos que justifican las probabilidades obtenidas experimentalmente. ¿Puedes encontrarlos?
3. Es interesante crear gráficas que muestren visualmente las diferencias entre los tres métodos. En este apartado queremos dibujar los puntos medios de las cuerdas para cada uno de los tres métodos. Debes modificar cada una de las funciones que calculaban la probabilidad para que devuelvan una lista de las coordenadas cartesianas de los puntos medios. Finalmente podemos usar la instrucción

```
plot(point(simulacion1\_pm(10000),rgbcolor=hue(1),size=1)).show(aspect_ratio=1)
```

donde `simulacion1_pm(10000)` es la función que produce la lista de coordenadas de los puntos medios para el primer método, y 10000 es el número de puntos que queremos dibujar.

Puedes ver una solución de este ejercicio en la hoja [11?-PROBA-geom.sws](#).

11.6. Ejercicios

Ejercicio 12.

Supongamos una moneda trucada con probabilidad $p = 0.1$ de obtener cara. Determinar la probabilidad de que sea necesario lanzarla 100 veces antes de obtener por primera vez una cara.

Ejercicio 13.

En teoría de números se utilizan en ocasiones argumentos probabilísticos, por ejemplo para demostrar teoremas de existencia. En este ejercicio calculamos la **probabilidad de que dos enteros elegidos al azar sean primos entre sí**.

Para dar sentido al problema suponemos inicialmente que los enteros pertenecen al intervalo $[2, N]$, de forma que podemos calcular experimentalmente la probabilidad de que un par de enteros del intervalo sean primos entre sí, y finalmente trataremos de estudiar el límite de esa probabilidad cuando N tiende a infinito.

Trataremos de calcular unas cuantas cifras decimales del límite que ya no cambien cuando incrementamos N , y podremos usar, como en [el ejercicio sobre la aproximación de Stirling](#), la [Enciclopedia de sucesiones de enteros](#).

Ejercicio 14.

Supongamos que para cubrir un puesto de trabajo hay n candidatos que deben ser entrevistados para decidir cuál de ellos obtendrá el puesto. Las condiciones detalladas del problema son las siguientes:

1. El número n de candidatos es conocido al comenzar el proceso de selección.
2. Si entrevistáramos a todos los candidatos los podríamos ordenar de mejor a peor sin empates.
3. Los candidatos son entrevistados de uno en uno y en orden aleatorio, con cada una de las $n!$ posibles ordenaciones elegida con probabilidad $1/n!$.
4. Después de cada entrevista el candidato es aceptado o rechazado, y esta decisión es irrevocable
5. La estrategia que utilizamos consiste en entrevistar a un cierto número r de candidatos y elegir el siguiente entrevistado que es mejor que los r . Se puede demostrar que esta estrategia es óptima.

El problema consiste en

1. Entender el valor de r , en función de n , que hace máxima la probabilidad de elegir al mejor candidato de los n .
2. Determinar el valor límite de esa probabilidad cuando n tiende a infinito.

Queremos resolver este problema experimentalmente con la ayuda de `find_fit` y de la [Enciclopedia de sucesiones de enteros](#).

Capítulo 12

Miscelánea

En este último capítulo, especie de cajón de sastre (¿cajón desastre?), se incluyen ejercicios sobre temas sueltos que no encajan en los previos pero que creemos interesantes. No suele ser posible cubrirlos todos durante el curso y, de hecho, esperamos ir añadiendo otros nuevos en cursos sucesivos y cada curso seleccionar algunos.

12.1. Autómatas celulares

Los autómatas celulares consisten en un tablero, una configuración inicial del tablero y unas reglas que determinan como se pasa de una configuración a la siguiente, es decir, determinan la *evolución temporal* del autómata. El tablero puede ser 1-dimensional, 2-dimensional, etc., y cada casilla, que debe estar coloreada para distinguirla de las vecinas, puede ser de un color elegido entre k posibles, aunque frecuentemente $k = 2$. Las reglas de evolución del autómata dependen, para cada casilla, de los colores que presentan las casillas vecinas.

El aspecto quizá más interesante en los autómatas celulares es que, aunque su evolución está totalmente determinada por el tablero inicial y las reglas de evolución, y por tanto es un sistema completamente determinista, puede generar comportamientos complejos que muestran cierta (pseudo-)aleatoriedad.

Otro aspecto interesante es que algunos autómatas celulares muestran un *comportamiento reversible*, en el sentido de que la situación en el momento actual no sólo determina el comportamiento futuro sino también el pasado. Como las leyes físicas

fundamentales son reversibles, el estudio de estos modelos merece la pena.

Puedes leer sobre la teoría de los autómatas celulares en [este artículo](#).

12.1.1. El Juego de la Vida

Se trata de un autómata celular bidimensional, que simula, hasta cierto punto, la evolución de una *colonia de bacterias*. Puedes verlo en funcionamiento pinchando en el enlace en la esquina superior izquierda de esta [página web](#).

También puedes leer sobre el juego y algunas de las configuraciones iniciales más interesantes en [este artículo de la Wikipedia](#).

1. Se trata de un juego con cero jugadores. El juego comienza en una configuración inicial y evoluciona solo.
2. Se juega en un tablero, como de ajedrez, infinito con casillas vivas, que serán las blancas, y casillas muertas las negras. La configuración inicial, que suele tener un número finito de casillas vivas, es arbitraria (no la del tablero de ajedrez), y en cada etapa del juego las casillas pueden permanecer en su estado o cambiar al otro de acuerdo con reglas precisas.
3. Las reglas que determinan el estado de una casilla en la etapa siguiente dependen del estado en esa etapa de las ocho casillas vecinas: dos en horizontal, dos en vertical y cuatro en las diagonales. Concretamente son las siguientes:
 - a) Una casilla viva con menos (estricto) de dos vecinos vivos muere, de *aburrimiento*, en la siguiente fase .
 - b) Una casilla viva con dos o tres vecinos vivos se mantiene viva en la siguiente fase.
 - c) Una casilla viva con más de tres vecinos vivos muere debido al *stress que le causa su intensa vida social*.
 - d) Una casilla muerta con exactamente tres vecinos vivos cambia a viva en la siguiente fase porque *los tres vecinos vivos (!entre los tres, dos sólo no pueden!) han procreado*.

NÓTESE, y es importante para entender el proceso, que la actualización entre una fase y la siguiente se hace *de golpe* en todo el tablero, es decir, si cuando pasamos de la etapa n a la $n + 1$ hemos cambiado una casilla de muerta a viva ese cambio no influye en el resultado final en la etapa $n + 1$ que sólo depende del estado final en la etapa n .

En particular, la observación anterior obliga a mantener en memoria el estado del tablero en la fase n mientras vamos actualizando un tablero inicialmente completamente muerto hasta obtener el estado $n + 1$. Podemos representar el estado del tablero en cada fase mediante una matriz de ceros y unos.

4. En la práctica hay que hacer que el tablero sea finito para poderlo manejar en el ordenador, y vamos a usar *condiciones de periodicidad en el borde*: cada casilla en el borde, por ejemplo en el lateral de la izquierda, tiene cinco vecinos hacia el interior del tablero y los otros tres son los que, pegados al lateral de la derecha, ocupan posiciones simétricas de la casilla en cuestión y sus dos vecinos en vertical.

Decimos que el juego se está desarrollando en un *tablero toroidal*.

5. La idea de este juego es, claramente, que las reglas permitan estudiar la evolución en el tiempo de una *colonia de casillas*, simulando organismos vivos, que *nacen, se reproducen y mueren*, pero también se autoorganizan en *organismos pluricelulares* que pueden incluso *desplazarse* por el tablero.
6. Sorprende la enorme riqueza de comportamientos que se ha obtenido, pero todavía sorprende más que el juego es una *máquina universal de Turing*, lo que significa, más o menos, que cualquier problema para el que existe una solución algorítmica (i.e. problema que se puede resolver en un ordenador) se puede resolver dentro del juego de la vida. Por ejemplo, es posible codificar dos enteros binarios dentro de una configuración inicial de forma que en un número finito de fases se alcance un estado, que ya no varía, y en el que se puede leer la suma o el producto de los enteros.

Ejercicio 1.

PROGRAMAR el juego de la vida usando estas instrucciones:

1. Conviene crear una función `vecinos((n,m))` que devuelva los ocho vecinos de cada casilla (n, m) teniendo en cuenta que el tablero es un toro.
2. Una función `siguiente(M)` que tome como argumento una matriz cuadrada M , $N \times N$ de ceros y unos, correspondiente a la fase N del tablero y devuelva la matriz correspondiente a la fase $n + 1$.
3. Una función `itera_hasta(siguiente,M,T)` que reciba como argumentos la función `siguiente`, la matriz del estado inicial del tablero M y el número de iteraciones T , y devuelva una lista con entradas los primeros T estados del tablero.
4. Representación gráfica:

```
v = map(matrix_plot,itera_hasta(siguiente,M,20))
animate(v).show(delay=50,iterations=10)
```

En esta hoja de Sage puedes ver una solución [121-MISCE-vida.sws](#).

12.1.2. Autómatas celulares 1-dimensionales

En este ejercicio vamos a estudiar una familia de autómatas celulares de dimensión uno. El *juego de la vida* es un autómata celular de dimensión dos y, por tanto, algo más complicado que los que veremos en este ejercicio. El dato principal es una lista de ceros y unos que representará el estado inicial del autómata. En el juego de la vida, en lugar de una lista como estado inicial usamos una matriz.

El estudio sistemático, mediante ordenador, de los autómatas celulares fue comenzado por Stephen Wolfram¹, creador del programa de cálculo simbólico, rival de Sage, Mathematica. Wolfram ha escrito un tocho de unas 1200 páginas, *A new kind of science*, que aunque un poco megalomaniaco a veces, es, sin embargo, bastante interesante. La familia de autómatas considerada en este ejercicio fué definida y estudiada por él en los 80.

Ejercicio 2.

1. Define una función de Sage `vecinos(k,L)` que para cada entero k entre 0 y $\text{len}(L) - 1$ devuelva la 3-upla formada por el elemento de L anterior a $L[k]$, el elemento $L[k]$ y el siguiente a $L[k]$, teniendo en cuenta que el siguiente al último debe ser el primero y el anterior al primero debe ser el último.
2. Consideramos la lista de las 8 3-uplas de ceros y unos que podemos formar:

$$T = [(0, 0, 0), (1, 0, 0), (0, 1, 0), (1, 1, 0), (0, 0, 1), (1, 0, 1), (0, 1, 1), (1, 1, 1)].$$

3. Cada entero $0 \leq k \leq 255$ se puede escribir en binario con 8 bits (rellenando con ceros cuando haga falta). Dado un tal entero k le podemos hacer corresponder, de esta manera, una lista $B(k)$ de 8 ceros o unos.
4. Define una función de Sage que reciba como argumento un entero $0 \leq k \leq 255$ y devuelva un diccionario $D(k)$ con claves las 3-uplas de la lista T y, para cada 3-upla tome valor el cero o uno que ocupa el mismo lugar en la lista $B(k)$ del apartado anterior. Observa que el orden en que hemos definido la lista T de 3-uplas influye en el diccionario obtenido.
5. Define una función de Sage `siguiente(L,k)` que reciba como argumentos una lista L de ceros o unos y un entero $0 \leq k \leq 255$ y devuelva otra lista de la misma longitud que L que en el lugar i tenga el valor que el diccionario $D(k)$ asigna a la 3-upla de vecinos de $L[i]$. Esta función es la que determina la evolución en el tiempo del autómata.

¹Según información obtenida en Internet, quizá no totalmente fiable, el personaje *Sheldon Cooper* en la serie *The big bang theory* está basado directamente en la vida real de Wolfram.

6. Define una función `evolucion(L,k,N)` que devuelva la lista de listas, de longitud $N+1$, que se obtiene al iterar, N veces, la función `siguiente` partiendo del valor dado de L , es decir, debe devolver la lista

$$[L, \text{siguiente}(L), \text{siguiente}(\text{siguiente}(L)), \dots, \text{siguiente}^N(L)].$$

7. Podemos transformar la lista devuelta por `evolucion(L,k,N)` en una matriz con $N+1$ filas y $\text{len}(L)$ columnas, y representar gráficamente la matriz usando `matrix_plot`. Obtener esas representaciones gráficas para los valores $k = 18, 30, 50, 110$, $N = 256$ y la lista L inicial formada por 128 ceros, un uno y otros 128 ceros. Observa las diferencias en las características de los gráficos obtenidos.

12.2. Percolación

Se llama *teoría de la percolación* a una serie de modelos matemáticos que tratan de captar los aspectos importantes de la situación real de un líquido que se filtra a través de un material poroso. La teoría forma parte de la teoría de la Probabilidad, ya que un ingrediente fundamental es una probabilidad p fijada, y la misma en todos los puntos, de que el líquido se filtre de un punto a un punto vecino.

Es interesante que la percolación, como modelo matemático, presenta *fenómenos críticos*. Se puede demostrar matemáticamente, pero también estudiar experimentalmente en el ordenador, que existe un valor crítico p_c de la probabilidad que define el modelo tal que si $p \leq p_c$ no se produce con casi total seguridad percolación hasta infinito, mientras que si $p > p_c$ se produce con casi total seguridad percolación hasta infinito. Esto hace que el estudio de la percolación sea, en cierta medida, semejante al estudio de los cambios de fase de la materia en física (congelación, evaporación, etc.), para los que también existen temperaturas críticas que separan los diferentes estados.

Puedes leer más sobre la percolación en [este artículo](#).

DESCRIPCIÓN DEL PROBLEMA:

1. Consideramos el retículo L formado por los puntos de coordenadas enteras, a los que llamamos nodos de L , en el plano $\mathbb{R} \times \mathbb{R}$ junto con los segmentos que unen cada uno de esos puntos a los adyacentes. Cada nodo (n, m) tiene cuatro adyacentes, o vecinos, $(n-1, m)$, $(n, m-1)$, $(n+1, m)$ y $(n, m+1)$.

2. Fijamos un número real $p \in (0, 1)$ y para cada uno de los cuatro vecinos del origen $(0, 0)$ lanzamos una moneda trucada con probabilidad p de obtener cara y, en caso de obtener cara, decimos que un líquido que se genera en el origen ha *percolado* al nodo en cuestión.
3. Este proceso se repite para cada nodo al que el líquido ha percolado, y se trata de analizar los valores de p para los que el líquido *percola hasta infinito*.
4. Se puede considerar que éste es un modelo de un líquido que se filtra a través de un sólido poroso, con *porosidad* controlada por la probabilidad p , y tratamos de averiguar si el líquido *transpasa* o no. Por supuesto es un modelo matemático muy idealizado y, por ejemplo, en *la realidad* el sólido no será infinito.
5. También puede verse como un modelo de la transmisión de una enfermedad contagiosa, que comienza en $(0, 0)$ y cuya probabilidad de contagio de una persona enferma a sus *próximas* es p .
6. ¿Se podría estudiar mediante un modelo similar la propagación de los incendios forestales? La dificultad radicaría en que la probabilidad de propagación de un punto a los próximos deberá depender de las condiciones locales (existencia de material combustible, viento en cada momento, orografía, etc.) cerca del punto, y no parece fácil determinarla.

Ejercicio 3. Programamos la percolación en un retículo de cuadrados según las siguientes indicaciones:

1. Debemos mantener dos listas que irán variando a lo largo del proceso, que podemos suponer que se desarrolla en instantes de tiempo sucesivos:
 - a) Una de todos los nodos a los que el líquido ha percolado hasta ese momento, a la que podemos llamar `nodos_visitados`.
 - b) Otra de los nodos alcanzados en el instante anterior, a la que podemos llamar `nodos_alcanzados`.
2. Inicialmente, `nodos_visitados=nodos_alcanzados=[(0,0)]`, pero, por ejemplo, puede ocurrir que en el siguiente instante tengamos `nodos_visitados=[(0,0), (1,0), (0,-1)]` y, por tanto, `nodos_alcanzados=[(1,0), (0,-1)]`.
3. Entonces, la moneda se lanza, en cada instante, desde cada uno de los nodos alcanzados en el instante previo y, según el resultado, se actualiza la otra lista añadiéndole los nodos a los que el líquido ha percolado y que no estaban ya en ella.
4. El proceso para naturalmente si en un instante no hay percolación desde ninguno de los nodos alcanzados, pero para valores de p altos debemos esperar que el líquido siga percolando indefinidamente y debemos parar el proceso nosotros.

5. Podemos pararlo limitando el número veces que se va a repetir el intento de percolar, o bien parando el programa cuando se obtengan puntos con coordenadas mayores que un límite prefijado. Es claro que no podemos determinar, usando el ordenador, si se va a producir percolación hasta infinito o no, pero variando estos límites podemos hacernos una muy buena idea.
6. Una variante del programa permitiría visualizar el resultado:
 - a) Definimos una matriz M , de tamaño $(2N + 1) \times (2N + 1)$ con filas y columnas indexadas por enteros entre 0 y $2N$ (ambos inclusive) que va a consistir únicamente de ceros o unos.
 - b) Cada vez que se visita un nodo (n, m) hacemos que la entrada $M[n + N, m + N]$ de la matriz sea un 1, y los nodos no visitados quedan representados por un cero en la matriz.
 - c) En este caso, para parar la percolación, no añadimos a la lista de nodos alcanzados aquellos nodos que, aunque el líquido hubiera percolado a ellos, estén fuera del cuadrado $[-N, N] \times [-N, N]$.
 - d) Podemos visualizar el resultado usando la instrucción `matrix_plot(M)`.

Una vez que hemos conseguido que el programa funcione, debemos estudiar, en función del valor de $0 < p < 1$, cuándo se produce percolación hasta infinito. Es claro que si p es pequeña no la habrá mientras que si es próxima a 1 debería producirse, ya que, para cada nodo alcanzado sorteamos 4 veces para ver si percola a alguno de los nodos vecinos. La probabilidad de que no percole a ninguno sabemos que vale $(1 - p)^4$, y sería muy pequeña.

Debemos esperar entonces que en algún valor p_c de p se produzca el cambio de comportamiento: para $p < p_c$ el líquido no percola hasta infinito mientras que para $p > p_c$ sí lo hace. Decimos que p_c es la *probabilidad crítica*, y nos interesa, en primer lugar, determinarla experimentalmente.

12.3. Ley de Benford

La *Ley de Benford* se originó como una observación empírica, con el siguiente contenido:

Decimos que una sucesión cumple la Ley de Benford si la frecuencia del dígito i como dígito dominante en los términos de la sucesión es, con muy buena aproximación, $B[i] := \log_{10}(1 + \frac{1}{i+1})$.

Puedes leer un poco más sobre la ley en [este artículo](#).

Ejercicio 4.

1. Define una función de Sage `frecuencia_fib(N)` que tenga como argumento un entero N y devuelva la lista de frecuencias de cada uno de los dígitos (1,2,3,4,5,6,7,8,9) como dígito dominante (i.e. el primero por la izquierda) de los números en la sucesión de Fibonacci que empezando en 1 tiene longitud N . Representa gráficamente las frecuencias obtenidas.
2. Denotemos por F la lista que devuelve `frecuencia_fib(20000)`. Compara, usando una función de Sage, los valores $F[i]$ con los números $B[i] := \log_{10}(1 + \frac{1}{i+1})$.
3. Modifica la función del apartado 1) para calcular las mismas frecuencias pero para la sucesión de potencias 2^j con j recorriendo los enteros en el intervalo $[1, N]$. ¿Qué observas al calcular las frecuencias con $N = 20000$?
4. Modifica la función del apartado 1) para calcular las mismas frecuencias pero para la sucesión de enteros en el intervalo $[1, N]$, y también para la sucesión de cuadrados j^2 con j recorriendo los enteros en el intervalo $[1, N]$. ¿Qué observas al calcular las frecuencias con $N = 20000$?
5. La Ley de Benford se cumple para muchas sucesiones generadas mediante una expresión matemática (no para todas), y para muchas sucesiones de datos numéricos obtenidos del mundo real, por ejemplo, mediante estadísticas. ¿Cuál puede ser la diferencia fundamental entre sucesiones que cumplen bien la Ley de Benford y las que la cumplen medio bien o no la cumplen?

12.4. Tratamiento de imágenes

Una aplicación interesante del Álgebra Lineal consiste en usar la descomposición en valores singulares (SVD) para la compresión de imágenes. Intentamos reducir el tamaño de un archivo que contiene una imagen sin una gran pérdida de calidad.

La descomposición en valores singulares de una matriz $m \times n$, con entradas reales, A consiste en la factorización

$$A = U \cdot \Lambda V^T,$$

con U y V matrices cuadradas ortogonales (i.e. con inversa igual a la transpuesta), y Λ una matriz $m \times n$ con ceros fuera de la diagonal principal. Los elementos no nulos de Λ son los llamados *valores singulares* de A , y son las raíces cuadradas positivas de los valores propios de la matriz simétrica $A \cdot A^T$, que por ser simétrica es diagonalizable.

Puedes leer sobre la descomposición y sus aplicaciones en [este artículo](#), y [en las notas](#) de Eugenio Hernández sobre el tema.

Además puedes consultar la hoja de Sage [125-MISCE-imagenes.sws](#), que contiene ejemplos en los que se aplica esta técnica.

12.5. Sistemas polinomiales de ecuaciones

Como bien sabemos, los sistemas lineales de ecuaciones, con coeficientes en un cuerpo, se pueden resolver siempre sin salir del cuerpo de coeficientes. El método estándar es la *reducción gaussiana+sustitución*, que se puede programar fácilmente y resuelve de manera exacta los sistemas lineales con coeficientes en los racionales o en un cuerpo finito².

Frecuentemente nos encontramos en la necesidad de resolver sistemas de ecuaciones no lineales, por ejemplo polinomiales o trascendentes (i.e. senos, cosenos, exponenciales, logaritmos, etc.) , y la triste realidad es que NO EXISTEN MÉTODOS GENERALES DE SOLUCIÓN. En cada caso tratamos de resolverlo usando algún *truco*, que puede servir en ese ejemplo y quizá en unos pocos similares.

Sin embargo, en el caso de sistemas de ecuaciones polinomiales en varias variables es posible adaptar la reducción gaussiana para aproximarnos de manera sistemática a algo parecido a una solución. De hecho, el método, que se conoce como de *bases de Gröbner*, utiliza ideas que proceden del algoritmo de reducción gaussiana junto con otras que proceden del algoritmo de Euclides para el máximo común divisor.

Supongamos dado un sistema de ecuaciones polinomiales

$$F_1(x_1, x_2, \dots, x_m) = 0$$

$$F_2(x_1, x_2, \dots, x_m) = 0$$

$$\vdots$$

$$F_n(x_1, x_2, \dots, x_m) = 0$$

con los F_i polinomios de grado k_i en las m variables. La teoría de bases de Gröbner nos permite obtener un sistema de ecuaciones con las mismas soluciones que el dado, pero con una estructura *escalonada*. Concretamente, si suponemos que el sistema (??) tiene un número finito de soluciones, obtenemos un sistema

²Es cierto que podemos encontrar las soluciones en un cuerpo finito de un sistema de ecuaciones simplemente probando todas las posibles soluciones, pero si el cuerpo finito es grande, o el número de variables enorme, este método de fuerza bruta no será el mejor posible.

$$\begin{aligned}
G_1(x_1, x_2, \dots, x_m) &= 0 \\
G_2(x_1, x_2, \dots, x_m) &= 0 \\
&\vdots \\
G_N(x_m) &= 0
\end{aligned}$$

que tiene el mismo conjunto de soluciones que (??), pero una ecuación, la última, con una sólo incógnita. En las otras ecuaciones pueden no aparecer algunas de las variables, pero si SUPIÉRAMOS RESOLVER la ecuación polinomial con una sola incógnita $G_N(x_m) = 0$, podríamos sustituir sus soluciones en las otras ecuaciones y obtendríamos sistemas de ecuaciones con $m - 1$ incógnitas. De esta manera podríamos, en principio, resolver el sistema original.

El problema es que tampoco sabemos resolver ecuaciones polinomiales en una única variable: sabemos que una ecuación polinomial en una variable de grado k tiene k raíces complejas, contadas con su multiplicidad, pero únicamente podríamos, en general, calcular sus valores aproximados con un cierto número de dígitos prefijado. El problema es siempre la continuidad de los números reales, necesitamos infinitos decimales para determinar un real no racional, frente a la finitud de la memoria RAM del ordenador.

12.5.1. Bases de Gröbner en Sage

Usamos como cuerpo de coeficientes el de los números racionales.

1. Comenzamos definiendo el anillo de polinomios en varias variables en el que vamos a operar:

```
R = PolynomialRing(QQ,3,'xyz',order='lp')
```

R es el anillo de polinomios con coeficientes racionales en tres variables x, y, z . El último parámetro indica cómo se ordenan los monomios, y como no entramos en los detalles de la teoría no nos interesa.

2. Definimos el sistema de ecuaciones:

```
x,y,z = R.gens()
J = (2*x^2+y^2+z^2,x^2+2*y^2+z^2,x^2+y^2+2*z^2)*R
```

que corresponde al sistema de ecuaciones

$$2x^2 + y^2 + z^2 = 0,$$

$$x^2 + 2y^2 + z^2 = 0,$$

$$x^2 + y^2 + 2z^2 = 0.$$

Este sistema puede ser resuelto fácilmente *a mano*, pero en general los sistemas de ecuaciones cuadráticas son difíciles, y se puede demostrar que, en un cierto sentido, contienen toda la dificultad de los sistemas de ecuaciones de grados arbitrarios.

3. Cálculo de la base de Gröbner:

```
B = J.groebner_basis();B
```

```
[x^2, y^2, z^2]
```

Vemos que la única solución del sistema es $(0, 0, 0)$ con una cierta multiplicidad que podríamos definir y calcular usando la base de Gröbner.

4. Otro ejemplo:

```
x,y,z = R.gens()
```

```
J2 = (2*x^2+y^2+z^2+y*z,x^2+2*y^2+z^2+x*z,2*x^2+y^2+2*z^2+x*y)*R
```

```
B2 = J2.groebner_basis();B2
```

que corresponde al sistema de ecuaciones, algo más complicado,

$$2x^2 + y^2 + z^2 + yz = 0,$$

$$x^2 + 2y^2 + z^2 + xz = 0,$$

$$x^2 + y^2 + 2z^2 + xy = 0.$$

La base de Gröbner calculada es

$$\left[x^2 + \frac{1}{2}y^2 + \frac{1}{2}yz + \frac{1}{2}z^2, xy - yz + z^2, xz + \frac{3}{2}y^2 - \frac{1}{2}yz + \frac{1}{2}z^2, y^3 - \frac{4}{5}z^3, y^2z - \frac{1}{10}z^3, yz^2 + \frac{1}{10}z^3, z^4 \right]$$

y vemos que tiene una ecuación en una única variable $z^4 = 0$, con solución $z = 0$. En este caso es fácil, partiendo de que toda solución tiene la coordenada z igual a cero, terminar de calcular todas las soluciones.

5. Puede ocurrir que la base de Gröbner calculada no tenga ninguna ecuación en una sola variable. Se demuestra que esto únicamente puede ocurrir si el sistema original tiene infinitas soluciones complejas. Por ejemplo, si la base de Gröbner contiene un polinomio en dos variables $G_N(x_{m-1}, x_m) = 0$ basta ver que elegida una de las infinitas soluciones complejas de esta ecuación en dos variables, que forman lo que llamamos *una curva algebraica plana con ecuación* $G_N(x_{m-1}, x_m) = 0$, da lugar a una solución del sistema original en la que las últimas dos coordenadas vienen dadas por las coordenadas de la solución elegida de la ecuación en dos variables.

En este caso diríamos que la solución del sistema de ecuaciones original es una curva.

6. La teoría de las bases de Gröbner no sólo permite aproximarnos a la resolución de sistemas de ecuaciones polinomiales de grados arbitrarios, sino que es la base para los cálculos efectivos, con ordenador, en Geometría algebraica que es el estudio de conjuntos, curvas superficies, etc., dados como soluciones de sistemas de ecuaciones polinomiales.

12.6. Ejercicios

Ejercicio 5.

En este ejercicio vamos a simular un acuario de acuerdo a las siguientes reglas:

1. *El acuario es un cubo $[0, N] \times [0, N] \times [0, N]$. Cada pez puede moverse en cada instante de tiempo incrementando, o decrementando, una unidad una sólo de las coordenadas del punto, con coordenadas enteras, que ocupa. Si el punto tiene una de las coordenadas 0 ó N el pez sólo puede moverse hacia el interior del acuario. Los movimientos de cada pez son aleatorios y equiprobables entre todas las posibilidades de movimiento que tiene. En lugar de referirnos a puntos con coordenadas enteras los llamaremos *celdas*, y los pensamos como pequeños cubitos que dividen el acuario. Cada celda tiene 6 celdas vecinas, que son a las que se pueden desplazar los peces.*

2. Hay dos tipos de peces, A y B con A un pez grande que come peces chicos de tipo B . Los peces tienen sexo M o bien H .
3. Los peces pequeños comen *placton*. El plancton no se mueve y cada t_P períodos de tiempo coloniza cada una de las celdas vecinas con probabilidad p_1 . Es decir, para cada una de las 6 celdas vecinas pondremos una unidad de plancton si la moneda trucada ha salido 1. Una celda puede tener varias unidades de plancton hasta llegar a un límite de saturación S .
4. Un pez pequeño que llega a una celda que tiene plancton se queda en ella hasta que se acaba. En cada período de tiempo cada pez pequeño come, si en la celda hay, una unidad de plancton. Si en una celda que tiene plancton hay más peces pequeños que plancton se sortea qué peces comen.
5. Un pez pequeño que no ha comido durante t_B unidades de tiempo muere (inanición). Un pez grande que no ha comido durante t_A unidades de tiempo muere (inanición). Un pez pequeño vive durante T_B unidades de tiempo (muerte natural) y uno grande durante T_A (muerte natural).
6. Cuando un pez grande llega a una celda en la que hay peces pequeños se puede comer uno con probabilidad $p_{A,B}$ y si no se lo come el pez pequeño puede moverse, en el siguiente momento de tiempo, a otra celda. Un pez grande que llega a una celda en la que hay peces pequeños se queda en ella hasta que ya no hay. En cada período de tiempo un pez grande sólo se puede comer a uno chico. Si en una celda hay más peces grandes que peces pequeños se sortea cuáles de los grandes van a tener la posibilidad de comer.
7. Cuando dos peces del mismo tipo y distinto sexo llegan a una misma celda producen descendencia de tamaños D_A o D_B , la mitad de cada sexo. Si el pez está al borde de la inanición prefiere comer, si es posible, antes de reproducirse.
8. En el acuario hay inicialmente n_A peces de tipo A , n_B peces del tipo B y n_P unidades de plancton. La distribución inicial se puede hacer aleatoriamente y el número de peces de cada sexo debe ser igual.
9. Si en algún momento del desarrollo del programa necesitas más reglas puedes añadirlas, pero debes formularlas de la forma más clara posible. El modelo se puede complicar todavía más, por ejemplo debe ser interesante permitir que los peces actúen de manera *inteligente* moviéndose no a una celda vecina aleatoriamente sino a la que contenga más alimento (o mayor posibilidad de reproducirse).

Una vez programado el modelo hay que estudiar el comportamiento a largo plazo dependiendo de los parámetros, y sobre todo buscar valores iniciales que no producen la extinción.

Ejercicio 6. En este ejercicio definimos un autómata celular que simula avalanchas, por ejemplo de nieve o de arena. Las reglas son como sigue:

1. El estado del sistema en el momento t , $t = 0, 1, 2, 3, \dots$, se representa como una matriz $N \times N$, a la que llamamos, por ejemplo, $M(t)$. Los dos índices de la matriz varían entre 0 y $N - 1$, con entradas $M(t)_{ij}$ enteros no negativos. Esta matriz representa un tablero con N^2 casillas, y el entero $M(t)_{ij}$ es el número de granos de arena en la casilla de la fila i -ésima y columna j -ésima.
 2. En este tablero cada casilla, por ejemplo (i, j) , menos las del borde, tiene 4 casillas vecinas: $(i + 1, j)$, $(i, j + 1)$, $(i - 1, j)$, $(i, j - 1)$.
 3. Hemos definido un entero K al que llamamos *valor crítico*. Supondremos que $K \geq 4$.
 4. Para pasar de $M(t)$ a $M(t + 1)$: Se revisa cada una de las casillas y si $M(t)_{ij} > K$ hacemos $M(t + 1)_{ij} = M(t)_{ij} - 4$ e incrementamos en una unidad el entero de cada uno de los vecinos de nuestra casilla. Hay que tener presente que si la casilla es del borde tiene menos de 4 vecinos, y hay que tratar ese caso con cuidado. Si $M(t)_{ij} \leq K$ hacemos $M(t + 1)_{ij} = M(t)_{ij}$.
 5. Generamos el estado inicial $M(0)$ considerando un tablero que tiene en cada casilla un entero aleatorio del intervalo $[K + 1, 2K]$ y le aplicamos el procedimiento del apartado anterior, de forma reiterada, hasta que la matriz ya no cambie. La matriz obtenida, que tiene en todas sus casillas un entero menor o igual a K , será el estado inicial.
 6. Para producir avalanchas primero generamos un estado inicial $M(0)$ de la matriz, elegimos una casilla al azar y le añadimos un cierto número de granos de arena, por ejemplo I , y, finalmente, iteramos el procedimiento del apartado 4) hasta que la matriz ya no varíe.
1. DEFINE una función *evolución*(M, N, K), con M una matriz $N \times N$ con entradas enteros no negativos y K un entero, el valor crítico. La función debe implementar el procedimiento descrito en el apartado 4, y, suponiendo que M es el estado del tablero en el momento t , devolver el tablero en el $t + 1$.

2. DEFINE una función `ini(N,K)`, que implemente el procedimiento descrito en el apartado 5, y, por tanto, devuelva una matriz que tiene en todas sus casillas un entero menor o igual a K . COMPRUEBA que la matriz resultante de ejecutar `ini(10,5)` tiene todas sus entradas en el intervalo $[0, 5]$.
3. DEFINE una función `avalancha(N,K,I)`, que implemente el procedimiento descrito en el apartado 6 y devuelva el estado final de la matriz y la diferencia entre el estado inicial y el final. Esa diferencia nos permitirá visualizar el lugar en el que se ha producido la avalancha y su intensidad.

TRATA de obtener, variando los parámetros N , K e I , avalanchas grandes.

Apéndice E

Recursos

E.1. Bibliografía

1. DOCUMENTACIÓN PRODUCIDA POR SAGEMATH: Los desarrolladores de Sage producen y mantienen actualizada una cantidad inmensa de documentación sobre el sistema. Puede verse completa en [este enlace](#).

Hemos seleccionado y la que nos parece más interesante, de forma que no es necesario decargarla. Hay zonas de estos documentos que todavía no han sido escritas, es decir, la documentación parece estar siempre en desarrollo.

Por otra parte, parte de estos documentos se refieren a partes de las matemáticas mucho más avanzadas, y que, por supuesto, no nos conciernen en este curso.

- a) En primer lugar tenemos el [tutorial](#) de Sage, del que los tres primeros capítulos son lectura muy recomendable. Su contenido se puede solapar en parte con el del comienzo de estas notas, pero encontraréis allí multitud de ejemplos cortos que ayudan a comenzar con Sage.
- b) El [documento sage-power.pdf](#) contiene información más avanzada sobre el uso de Sage.
- c) Este otro [documento](#) contiene información sobre el uso de Sage en campos concretos de las matemáticas. En particular, pueden ser útiles la secciones 6.1.7 (teoría de números y criptografía), 6.1.1 (teoría de grafos) y el capítulo 5 (técnicas de programación).

- d) En este [documento](#) se explica cómo funcionan muchos de los objetos abstractos, grupos, anillos, etc., que se pueden construir y manipular en Sage.

2. OTROS:

- a) Un [libro](#) interesante de introducción a Sage. “Desgraciadamente” está en francés, pero los ejemplos de código sirven.
- b) Otro [texto](#) con aplicaciones, sobre todo, a las matemáticas de la etapa preuniversitaria.
- c) Este [texto](#) se centra en las matemáticas de los primeros cursos de Universidad.

3. PYTHON:

Sage incluye varios sistemas preexistentes de cálculo simbólico y el *pegamento* que los hace funcionar juntos es Python. Cuando queremos sacarle partido a Sage necesitamos utilizar Python para definir nuestras propias funciones, que frecuentemente incluyen funciones de Sage dentro. Python es entonces también el *pegamento* que nos permite obtener de Sage respuestas a problemas que no estaban ya preprogramados.

- a) En primer lugar tenemos aquí un [tutorial](#) de Python. Utiliza la versión 3 del lenguaje, mientras que Sage todavía usa la 2.7.
- b) [El tutorial](#) escrito por el autor, Guido van Rossum, del lenguaje. También para la versión 3 y bastante avanzado.
- c) [Una muy buena introducción](#) al lenguaje. Hay una [versión](#) del texto que utiliza Python 3. Allen B. Downey es también el autor de este [libro](#), en el que se aplica Python a diversos problemas, como los autómatas celulares, relacionados con la *teoría de la complejidad*. Este último texto contiene muchas ideas y ejercicios interesantes, y lo hemos usado para preparar la parte final del curso.

4. BIBLIOGRAFÍA POR MATERIAS:

- a) Un [estupendo resumen](#) del uso de Sage en matemáticas, sobre todo en cálculo y representaciones gráficas.
- b) [Un curso](#) de cálculo diferencial que utiliza Sage masivamente. No contiene cálculo integral.
- c) En este [sitio web](#) (parte de la documentación oficial de Sage) puede verse otra introducción al estudio del cálculo diferencial usando Sage.
- d) [Un curso tradicional](#) de Álgebra lineal que contiene una [extensión](#) sobre el uso de Sage para resolver problemas.

- e) Un **texto** bastante completo de criptografía que utiliza Sage para realizar los cálculos.
- f) Un **curso** muy popular, llamado CHANCE, de introducción a la teoría de de probabilidades. Los primeros capítulos pueden ser de alguna utilidad cuando lleguemos al capítulo 11.

E.2. Otros

1. Ya se **mencionaron los chuleteros** de Sage como forma rápida de acceder a la sintaxis de las principales instrucciones preprogramadas.
2. También se indicó en el prólogo de las notas que este curso se basa, en gran medida, en sus primeras versiones que montó Pablo Angulo. Puedes ver el curso original en **este enlace**.
3. El **sitio web rosettacode.org** es especialmente útil cuando se sabe ya programar en algún lenguaje pero se pretende aprender uno nuevo. Consiste en una serie grande de problemas resueltos en casi todos los lenguajes disponibles, en particular, casi todos están resueltos en los más populares como *C* y Python.

A fin de cuentas, casi todos los lenguajes tienen bucles **for** y **while**, bloques **if** y recursión. Hay entre ellos pequeñas diferencias en la sintaxis y en la forma en que se manejan las estructuras de datos.

4. En el **sitio web projecteuler** pueden encontrarse los enunciados de casi 500 problemas de programación. Dándose de alta en el sitio es posible ver las soluciones propuestas para cada uno de ellos.

Tal como están enunciados muchos de ellos son difíciles, ya que no se trata únicamente de escribir código que, en principio, calcule lo que se pide, sino que debe ejecutar el código en un ordenador normal de sobremesa, en un tiempo razonable, para valores de los parámetros muy altos. Es decir, lo que piden esos ejercicios es que se resuelva el problema mediante un código correcto y muy eficiente.