unir
LA UNIVERSIDAD DE INTERNET

FORMACIÓN PROFESIONAL

UNIDAD FORMATIVA 2

Construcción de software

Construcción de SW

Índice

Objetivos

Al terminar esta sección:

- Se habrán revisado algunos de los lenguajes que consideramos importante conocer y utilizar correctamente.
- Estaremos familiarizados con la teoría de la complejidad, aplicada al estudio de ciertos algoritmos.

Construcción de software

En esta unidad repasamos los pilares más básicos de la ingeniería del software y la programación, que nos van a valer para tener una mejor base respecto a la que evaluar, y ser competentes, en el campo de la construcción de software.

En nuestro día a día como DevOps nos podemos encontrar con muchos, muy diferentes escenarios, que pondrán a prueba nuestras capacidades como técnicos:

- Desarrollo de software de aplicaciones u operativa de operaciones.
- Mantenimiento de software creado por otros equipos.
- Revisión de software de otros equipos/compañeros.

El factor común en todos los casos es, sin duda, que se requieren conocimientos moderados o avanzados de desarrollo y mantenimiento de software.

El objetivo de esta lección no es convertirnos en expertos de la noche a la mañana, nada más lejos: la creación de software **es un arte** y, como tal, solo se mejora con la práctica constante y buscando retos que pongan a prueba nuestras capacidades.

Lo que buscamos es **repasar los lenguajes de programación**: sus tipos y algunas de las características de los que consideremos más apropiados. A continuación, pondremos en práctica lo que hemos aprendido repasando el concepto de **algoritmo** y algunos ejemplos.

Con esto, se persigue que el alumno adquiera cierta soltura en programación, que se refinará en la unidad siguiente (Testing).

Tipos de lenguajes

Existen muchas clasificaciones posibles entre los lenguajes de programación. La más común está entre lenguajes de **alto nivel** y lenguajes de **bajo nivel**: ensamblador y lenguaje de máquina.

Los lenguajes de bajo nivel están totalmente fuera del alcance de este curso, por lo que nos centraremos en buscar cómo clasificar los lenguajes de **alto nivel.**

Modo de ejecución

Se puede hablar de lenguajes **compilados** (Java, C, Go...) o **interpretados** (Bash, Python, JavaScript, R, PHP…).

Compilados

Los lenguajes compilados necesitan que su código fuente sea traducido (compilado) a lenguaje máquina: el lenguaje de más bajo nivel posible, que está optimizado para ejecutarse sobre cada sistema.

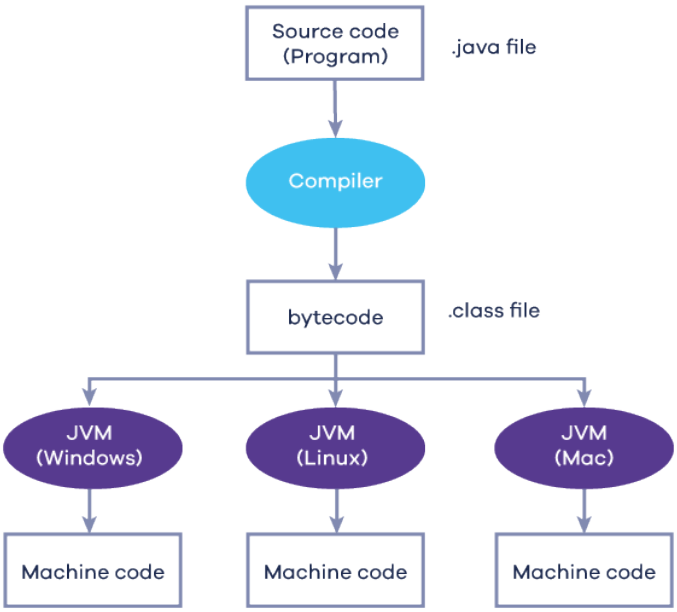
De esto se encargan los compiladores, herramientas muy avanzadas que, además, realizan multitud de chequeos por nosotros y nos avisan de errores de todo tipo que hacen que nuestro programa no tenga sentido:

- Tipos inválidos.
- Errores de sintaxis.
- Posibles errores en el tiempo de ejecución que se puedan prever.

Los compiladores son piezas de software que están muy atadas al sistema donde los ejecutamos, esto significa que el código, una vez compilado, será **dependiente de la plataforma**. Si queremos tener versiones específicas para Windows, Mac OSX, Debian Linux, RHEL…, tendremos que compilarlo por separado (y teniendo en cuenta sus diferencias).

El caso Java

En el caso de Java, no se traduce directamente a lenguaje máquina, sino a un formato a medio camino llamado bytecodes. Estos bytecodes se ejecutan sobre la **Java Virtual Machine** (JVM), que es un software independiente que corre sobre diferentes sistemas operativos y permite ejecutar el mismo código en diferentes sistemas **con un único bytecode.** Por tanto, usando Java se consigue código **independiente de la plataforma.**



Ejemplo:

Si tenemos el fichero 'ejemplo.java':

```
class Ejemplo {
  public static void main(String[] args) {
    for (int i=0 ; i<args[0].parseInt(); i++) {
      System.out.println(i);
    }
  }
```

Cuando lo intentamos compilar, 'javac' nos dice que tenemos un error:

```
$ javac ejemplo.java
ejemplo.java:3: error: cannot find symbol
    for (int i=0 ; i<args[0].parseInt(); i++) {
                     ^
  symbol:   method parseInt()
  location: class String
1 error
```

El compilador nos **avisa** de que la clase string no tiene ningún método *parseInt*, por lo que debemos arreglarlo y volver a probar.

Reto:

Haz funcionar este programa y consigue que cuente hasta el número que se le pase como argumento.

Interpretados

En los lenguajes interpretados no existe (o no es obligatorio) la fase de validación y compilación, y el código se ejecuta directamente desde las fuentes.

Esto **no significa** que no se pueda o se deba validar el código fuente: existen multitud de herramientas para garantizar que lo escrito es formalmente correcto, aunque es difícil conseguir el mismo nivel de garantía que en el caso de los lenguajes compilados.

1. **Testing**: una batería de test adecuada nos puede garantizar que el código haga lo que tiene que hacer, ante una entrada determinada
2. **Linting**: el proceso de *linting* se asegura, en base a una serie de reglas, que el código sigue unos estándares de programación y que no existen aparentes errores de sintaxis.
3. **Tipado estático**: en lenguajes como Python o Typescript se puede declarar el tipo de las variables, para que procesos como un *linter* o el motor de un IDE puedan detectar inconsistencias entre las asignaciones o parámetros de una función respecto a lo declarado.

Una de las grandes ventajas de estos lenguajes es que el proceso de prototipado y pruebas es muy rápido, al no precisar de compilaciones ni ser tan estrictos con la sintaxis… Aun así, no nos libramos de la gestión de dependencias, algo que nada tiene que ver con el tipo del lenguaje.

Paradigmas de programación

Y por último, existen una serie de **paradigmas**, y los diferentes lenguajes caen en uno o más de ellos. Los más conocidos son los imperativos y declarativos. Existen muchos otros tipos (Wikipedia), pero los más habituales en el mundo DevOps (y en ingeniería del software) son estos dos, con sus derivados.

Imperativos

Lenguajes donde el programador le dice a la máquina qué debe hacer mediante **instrucciones** que cambian su **estado.** Es un paradigma de programación que se asemeja a cómo funcionan los ordenadores a bajo nivel: asignaciones de variables a posiciones de memoria y ejecución secuencial de operaciones.

En cualquier caso, un lenguaje imperativo **puro**, a alto nivel, no suele ser tan práctico y es por eso que han surgido derivados que los hacen mucho más potentes, aunque el concepto subyacente sea el mismo:

- Procedimentales, lenguajes imperativos donde prima la **modularización** y **reutilización** de bloques de código (módulos, procedimientos o funciones).
- Orientados a objetos, que se basan en el concepto de **objetos**, estructuras que almacenan datos y estado.

Aunque se pueden encontrar ejemplos 'puros', la verdad es que la mayoría de los lenguajes usuales se pueden considerar 'procedimentales orientados a objetos'.

Ejemplos:

```
# rectangulo.py
class Rectangulo:
  def __init__(self, altura, base):
    self.__altura = altura
    self.__base = base


rectangulo1 = Rectangulo(10, 20)
rectangulo2 = Rectangulo(20, 20)
```

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
let p = new Rectangle(10, 20);
console.log(p);
```

Declarativos

Definidos a grandes rasgos como 'lenguajes que no son declarativos', son aquellos en los que se busca decirle a la máquina **qué** debe hacer, sin especificar el **cómo**. Son conocidos porque no guardan el estado de variables, eliminando la mutabilidad que suele caracterizar a los declarativos.

El derivado más conocido es, sin duda, **la programación funcional**. Este paradigma, donde las **funciones** se pueden componer, usar como valores de retorno y hasta pasar como argumento, han experimentado un aumento muy significativo de popularidad gracias a:

- Lenguajes funcionales puros como Elixir, Haskell o Clojure.
- Características funcionales incluidas en otros lenguajes, como la programación funcional de JavaScript, los streams de Java 8 o el soporte funcional de Scala.

Ejemplos:

```
new Random().ints().limit(100).filter( i -> i < 30 ).map( i -> i^2 ).sum();
```

```
Array.from({ length: 100 }).map( x => Math.random()*100 ).filter( x => x < 30 ).reduce( (a,b) => a + b )
```

Lenguajes recomendados

Con el objetivo de que el desarrollo de retos y prácticas sea lo más uniforme posible y consistente, con los ejemplos que se verán a lo largo de las diferentes asignaturas nos centraremos en los siguientes lenguajes de programación (entre paréntesis, versión mínima recomendada):

- Java (1.8, equivalente a Java 8).
- Python (3.x).

Mención especial merece Bash (v4), como lenguaje de scripting que debemos conocer y manejar también. No obstante, nos centraremos más adelante en él.

> ¿Qué significa la versión mínima?
>
> La evolución a lo largo del tiempo de los lenguajes de programación es algo a lo que tenemos que prestar atención, siempre. En especial, cuando el cambio se produce en una versión *major*, lo que en versionado semántico se entiende como que no se garantiza la retrocompatibilidad: un ejemplo es el paso de Python 2 a 3.

Pasamos a listar las razones por las que estos dos lenguajes son en los que merece la pena hacer más incidencia durante el curso.

Java

1. **Independiente de la plataforma**: gracias a la JVM, el código fuente de Java se puede compilar y ejecutar en cualquier sistema mientras tenga instalada una JVM.
2. **Conocido**: al ser un lenguaje muy extendido, no es difícil encontrar recursos online o profesionales con alto grado de pericia en el lenguaje.
3. **Sencillo**: es un lenguaje **relativamente** sencillo, muchos hemos empezado a programar en Java para aprender programación orientada a objetos, o incluso los fundamentos de la programación procedimental.
4. **Orientado a objetos**: es uno de sus fuertes, sin duda, y nos facilita hacer un buen diseño de una arquitectura si pensamos adecuadamente lo que queremos modelar.
5. **Alta presencia en la industria**: una gran cantidad de frameworks de alto nivel están hechos en Java o alguno de sus derivados (Kotlin, Scala, Groovy…).

En el mundo DevOps, ser un experto en Java nos permite de forma inmediata:

- Poder dar soporte a equipos que trabajan en algunos de los frameworks web más famosos que existen, como:
  - Spring.
  - Play!.
  - Struts.
- También es el lenguaje de las aplicaciones Android, aunque se esté viendo desplazado por Kotlin.

- Los microservicios Java son una manera muy común de **romper monolitos**: aplicaciones muy antiguas que han crecido a un tamaño desmesurado y cuya evolución y mantenimiento se han tornado demasiado costosos.

Ejemplo de servicio web Java

Se expone a continuación un servicio web sencillo, muy sencillo. En concreto, se busca huir de los complejos (aunque potentes) frameworks como los mencionados arriba, para que el alumno tenga un entorno en el que realizar pruebas o un ejemplo que ir iterando en las **sucesivas prácticas de la asignatura.**

```
import java.io.*;
import java.net.InetSocketAddress;
import java.util.stream.Collectors;

import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpServer;


class Server {
  public static void main(String[] args) throws IOException {
    int port = 9000;
    HttpServer server = HttpServer.create(new InetSocketAddress(port), 0);

    System.out.println("server started at " + port);
    server.createContext("/", new MyHandler());
    server.setExecutor(null);
    server.start();
  }

  static class MyHandler implements HttpHandler {
    @Override
    public void handle(HttpExchange t) throws IOException {
      String result = new BufferedReader(new InputStreamReader(t.getRequestBody()))
            .lines().parallel().collect(Collectors.joining("\n"));

      String response = "This is the response: " + result;
      t.sendResponseHeaders(200, response.length());
      OutputStream os = t.getResponseBody();
      os.write(response.getBytes());
      os.close();
    }
  }
}
```

Para probar, basta compilar el programa, ejecutar con Java y acceder a la ruta http://127.0.0.1 en el navegador o enviar peticiones con o sin payload usando la herramienta curl.

```
javac server.java
java Server &
curl -XPOST 127.0.0.1/ -d "Hola mundo!!"
```

Recursos

- Ejemplos básicos y Manual (vía http://www.manualweb.net/).
- Codewars Java (muy recomendado hacer al menos uno).

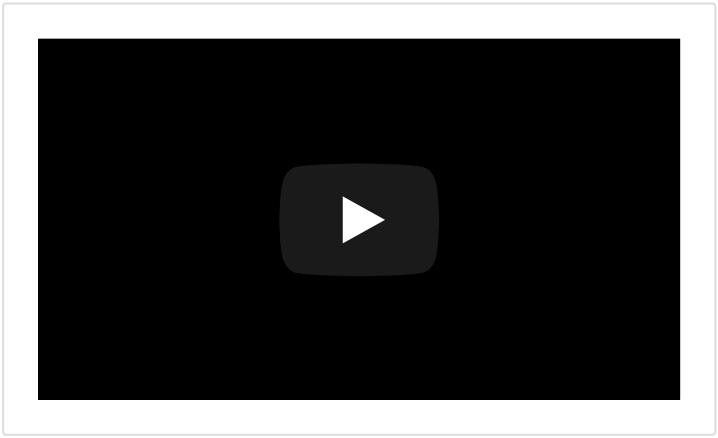
Python

Python comparte muchas de las cualidades de Java comentadas más arriba, aparte de la diferencia obvia en que este es un lenguaje interpretado y no tipado:

1. **Conocido.**
2. **Sencillo**: es un lenguaje relativamente sencillo también. Su curva de aprendizaje es muy baja, pero sin dejar de ser un lenguaje **muy potente**, y además con librerías de una utilidad increíble que lo convierten en una de las herramientas más usadas (sino la que más) en campos como data engineering.
3. **Orientado a objetos**: si bien no es exactamente como Java, sí que se pueden modelar entidades de todo tipo como elementos de bases de datos o actores en una aplicación web.
4. **Alta presencia en la data engineering**: no se concibe el análisis de datos sin Python, con herramientas como:
    - PySpark.
    - Pandas.
    - scikit-learn.
5. Potentes **web frameworks**, muy famosos por la facilidad con la que se pueden desplegar en cualquier proveedor o sistema de microservicios:
    - Django.
    - Tornado.
    - Flask.

En el mundo DevOps, ser un experto en Python nos permite, de forma inmediata:

- Dar soporte a equipos de data engineering, tanto en la elaboración y productivización de sus modelos de datos, como en la elaboración de sus pipelines de datos.
- Crear microservicios que implementan micro-APIs REST: al ser un lenguaje con un runtime tan pequeño, se puede crear imágenes muy pequeñas con un rendimiento muy potente.
- Generar utilidades de forma efectiva, ya que Python también es muy potente para el manejo de ficheros, llamadas al sistema operativo, etc.



Entornos virtuales y dependencia:

Un concepto muy importante con Python, que conviene tener claro desde el principio, es el de la **gestión de las dependencias**.

Python es un lenguaje que, de serie, no tiene todo lo necesario para efectuar su trabajo. Por tanto, más pronto que tarde vamos a necesitar instalar dependencias a nuestro proyecto, algo que en mundo Python se hace con la herramienta 'pip'.

```
# Lo instala bien, si, ¿pero donde?
pip install "requests==2.2.0"
```

El problema es que, si no hacemos nada al respecto, instalará ese paquete en una ubicación **global** del sistema, por lo que todos los scripts de Python que utilicen el mismo binario de Python tendrán que usar esa versión, algo que puede no ser deseable.

Pero eso no es lo más grave: desde un punto de vista DevOps, debemos garantizar que las aplicaciones pueden descargarse y ejecutarse en cualquier sitio y acabamos de instalar una dependencia de la misma en una ubicación global y sin apuntarla en ninguna parte.

Para esto, se han creado los entornos virtuales de Python ***(virtualenv)*** que, junto con unos ficheros de requerimientos que acompañan a cada proyecto Python, permiten que cada proyecto que tengamos tenga **su propio espacio de ejecución aislado** de los demás.

Esto, que aparentemente parece nada más que una conveniencia, es una de las razones por las que resulta tan sencillo convertir una aplicación Python en un microservicio que se ejecuta en un contenedor Docker.

El proceso es el siguiente:

1. Se especifican las dependencias en un fichero 'requirements.txt'.
2. Se crea un virtualenv **y se activa.**
3. Se instalan las dependencias ya con el virtualenv activo.

```
$ cat requirements.txt
requests==2.2.0
$ virtualenv venv
$ tree
.
├── requirements.txt
├── venv
└── source.py
$ source venv/bin/activate
$ which python
/home/user/apps/python/venv/bin/python
$  deactivate
$  which python  # no tiene por qué ser así
/home/user/.venvs/versions/3.8.2/envs/metrics-reports-3.8.2/bin/python
```

Cabe recordar que 'activar un virtualenv' no es más que la asignación de ciertas variables de entorno de la sesión que tenemos abierta para que los binarios de Python se lean de una carpeta determinada (en nuestro ejemplo, venv*).*

Ejemplo de servicio web Python:

Igual que para el caso Java, se expone un servicio web muy sencillo para que el alumno tenga un entorno en el que realizar pruebas o un ejemplo que ir iterando en las **sucesivas prácticas de la asignatura.**

El framework elegido es Flask y el ejemplo está tomado de su Quickstart:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

Reto

Ejecuta está pequeña aplicación dentro de su propio *virtualenv*.

Recursos online:

- Python en 10 minutos, de software Crafters.
- Tutorial de Python oficial para la última versión (3.10)
    - Recomendado hacer, al menos, los **tres primeros puntos.**
- Codewars Python (muy recomendado hacer al menos uno).

Otros lenguajes

Esto **no quiere decir** que no se deban conocer y manejar otros lenguajes de programación: esto **no trae más que ventajas** y en ocasiones habrá elecciones que resulten idóneas para ciertas operaciones. Ejemplos de lenguajes con aplicaciones muy diversas y con gran penetración o áreas de especialidad son:

- JavaScript
- C / C++
- C# / Swift
- Golang
- Scala
- Kotlin
- Groovy
- Elixir
- Clojure
- PHP
- Ruby
- Rust
- R

Reto

¿Echas en falta algún lenguaje en esa lista? Añádelo de forma justificada.

'Hello, World!'

En ocasiones, cuando se quiere evaluar un lenguaje, nos decidimos a escribir un sencillo programa que lo único que hace es escribir la cadena 'Hello, world!' o '¡Hola, mundo!'.

Esto nos permite evaluar la complejidad del manejo de cadenas, de la salida por consola/*standard output*, la compilación o ejecución de un programa sencillo…

Un ejemplo en Java:

```java
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

Y otro en Python:

```python
if __name__ == '__main__':
    print("Hello, World!")
```

Reto

Escribe 'Hello, World!' en un lenguaje que no hayas usado nunca.

Algoritmos

Un concepto que debemos conocer y manejar con cierta soltura son los diferentes algoritmos que se usan en el mundo de la programación.

Si bien es cierto que los lenguajes de más alto nivel incluyen eficientes librerías o funciones nativas de ordenación, filtrado, manejo de cadenas y, en general, de casi cualquiera de estos casos, su estudio nos aporta:

1. **Conocimiento**, pues sabremos que determinados problemas ya se han resuelto y para qué casos resultan óptimas determinadas soluciones.
2. **Ideas**, pues algunos de los más eficientes algoritmos tienen soluciones de lo más triviales o extremadamente ingeniosas.
3. **Práctica**, pues de la implementación de estos problemas aprenderemos detalles del lenguaje de programación elegido como manejo de estructuras, memoria, recursividad, etc.

¿Cómo evaluar un algoritmo?

En cualquier problema que tengamos que resolver por nuestra cuenta, **siempre** debemos ofrecer una solución óptima, o lo mejor posible dadas las circunstancias. A la hora de ofrecer una solución, tenemos que considerar:

- **Velocidad:** ¿cuánto tiempo necesita el algoritmo para devolver un resultado exacto o converger a uno suficientemente cercano a él?
- **Uso de recursos:** si usamos mucha memoria para una ordenación, o abusamos del trasiego a disco, quizá nuestra solución no escale adecuadamente cuando crezca el tamaño de los datos de entrada.

Es aquí donde debemos introducir un nuevo concepto: *la teoría de la complejidad.*

Teoría de la complejidad

Antes de poder abordar el estudio de algoritmos comunes, debemos conocer los conceptos más básicos de la teoría de la complejidad.

A muy grandes rasgos, esta teoría nos da las herramientas básicas para analizar la escalabilidad de un algoritmo conforme aumenta el tamaño de los datos de entrada. Esta escalabilidad se mide en dos dimensiones:

- **Complejidad temporal**

  La complejidad temporal hace referencia al tiempo de ejecución y a cómo crece en función del tamaño de la entrada del algoritmo. Es especialmente importante cuando tratamos de optimizar al máximo el rendimiento de nuestros algoritmos, ya que las ineficiencias en el código se traducirán en largas esperas.

- **Complejidad espacial**

  Esta complejidad hace referencia al consumo de espacio en disco (memoria o disco duro) de nuestro algoritmo. Es especialmente crítico cuando este recurso es limitado como, por ejemplo, en microservicios, o hardware de IoT.

Además de estas dos dimensiones, debemos considerar cuáles son las posibles condiciones en las que se ejecutará un algoritmo, para acotar la complejidad o dar un valor estadísticamente promedio:

1. Se dice que un algoritmo se ejecuta en el **mejor caso** cuando se dan las condiciones para que se ejecute en las menores iteraciones posibles, o con el menor coste posible en términos de tiempo/espacio.
2. De forma opuesta, un algoritmo se ejecuta en el **peor caso** cuando las condiciones son las peores posibles y el número posible de iteraciones y operaciones es **máximo**.
3. Entre ambos extremos está el **caso medio**, que mide el promedio del valor de la complejidad de un algoritmo cuando se consideran infinitas iteraciones con entradas aleatorias.

Para poder expresar esta complejidad en términos comparables, se ha creado una notación especial: Big-O Notation o simplemente, *notación O.*

Notación O

La notación O es una aproximación matemática que expresa el término de una función que más aporta al crecimiento de la misma. Veámoslo con un par de ejemplos.

Ejemplo 1:

La función $f(x) = 5x^4 + -x^2 + 1$ crece de forma cuadrática y el término que 'más rápido crece' es sin duda el primero, $5x^4$. Además, el coeficiente 5 es irrelevante para el valor que alcanza la función (5 es insignificante comparado con x^4 para valores elevados de x). Por tanto, diríamos que f(x) tiene complejidad x^4, o $O(x^4)$.

Ejemplo 2:

Procesar listas en un lenguaje de programación es un caso muy común en el que estimar la complejidad de nuestro algoritmo.

Imaginemos que queremos obtener los elementos de una lista de '**N**' enteros que están presentes en otra distinta con '**M**' elementos. Una implementación trivial pasaría por ir elemento a elemento, viendo si cada elemento de la lista 1 está en la lista 2.

1. **Mejor caso**: cuando la lista 2 tiene **un único elemento** el algoritmo se reduce a una comparación por elemento de lista 1, por lo que la complejidad se reduce a *O(N)* (obviamos el caso trivial de que esté vacía).

2. **Peor caso**: el peor caso sería aquel en que se tenga que recorrer la lista todas las veces, para valores muy grandes de N y M la complejidad será *O(N·M).*
3. **Caso promedio**: si asumimos que ambas listas tienen **aproximadamente** los mismos elementos, se podría aproximar la complejidad por $O(N^2)$.

Reto

¿Serías capaz de hacerlo mejor? ¿Existe alguna forma más sencilla de obtener el mismo resultado? ¿Qué complejidad tiene?

Recursividad

Se dice que un algoritmo o una función es **recursiva** si se llama a sí misma, con parámetros distintos, para resolver un problema.

Esta aparentemente sencilla definición esconde algunos algoritmos realmente ingeniosos, que solucionan ciertos problemas de una manera sencilla y elegante:

```
def fibonacci(n):
    if n==1 or n==0:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

```
public int factorial(int n) {
    if (n == 0) {
      return 1;
    }
    return n * factorial (n-1);
}
```

Estos son los ejemplos más comunes de funciones recursivas: la sucesión de Fibonacci y el cálculo del factorial. Pero muchas veces encontraremos que son la mejor solución a un problema, siempre que podamos dividirlo en trozos 'más pequeños': esto se conoce como la estrategia **divide y vencerás**.

Problemas de la recursividad

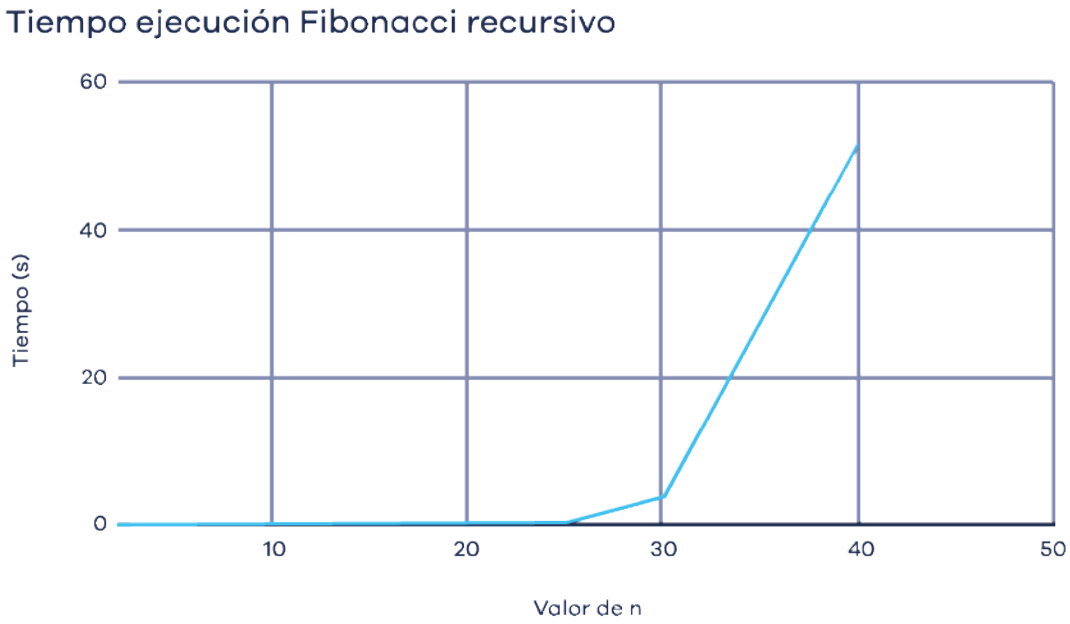
Si bien puede parecer tentador tratar de buscar soluciones recursivas a todos los problemas, y muchas veces es una buena alternativa, debemos tener en cuenta una serie de premisas:

1. **El consumo de memoria**: cada vez que llamamos a una función, se guarda el estado actual de la memoria para cuando regrese. Si este proceso no termina nunca, o itera demasiadas veces, nos quedaremos sin memoria, o incluso el sistema operativo nos cortará la ejecución por exceso de llamadas en la pila (*stack overflow)* antes de que podamos afectar al resto de procesos.

2. **La condición de salida:** relacionado con lo anterior, si no elegimos bien la condición de salida, o nos la 'saltamos', la función no acabará nunca.
3. **Malgasto de cómputo**: en una recursión múltiple, como en el caso de Fibonacci, se da el caso de que se calculará múltiples veces el mismo término de la sucesión. Si bien esto no es un problema para entradas pequeñas, el tiempo de ejecución crece de forma exponencial a medida que el número de iteraciones crece.

Hagamos una sencilla prueba, usando esta vez iPython y el magic command %%timeit:

```
In [18]: %%time
   ...: fibonacci(2)

CPU times: user 10 µs, sys: 1e+03 ns, total: 11 µs
Wall time: 15.3 µs
Out[18]: 1

In [19]: %%time
   ...: fibonacci(5)

CPU times: user 5 µs, sys: 0 ns, total: 5 µs
Wall time: 8.11 µs
Out[19]: 5
In [20]: %%time
   ...: fibonacci(20)

CPU times: user 3.58 ms, sys: 145 µs, total: 3.73 ms
Wall time: 4.18 ms
Out[20]: 6765

In [21]: %%time
   ...: fibonacci(25)
   ...:

CPU times: user 37.4 ms, sys: 2.97 ms, total: 40.4 ms
Wall time: 43.8 ms
Out[21]: 75025
In [22]: %%time
   ...: fibonacci(30)
   ...:
CPU times: user 379 ms, sys: 8.04 ms, total: 387 ms
Wall time: 420 ms
Out[22]: 832040
In [23]: %%time
   ...: fibonacci(40)
   ...:
   ...:
CPU times: user 50.6 s, sys: 1.11 s, total: 51.7 s
Wall time: 1min 7s
```



Reto

Para ver la diferencia en cantidad de código resultante, codifica ambos bloques de forma **no recursiva**: ¿cuál te parece más claro?

¿Cómo solucionarías el problema del crecimiento exponencial de la ejecución de Fibonacci?

Pseudocódigo

El pseudocódigo es una forma de expresar ideas, no exclusivamente algoritmos: podría ser el esbozo en papel de una solución o una formulación matemática rápida, o incluso una implementación de un problema en un lenguaje de programación inventado.

La idea tras el pseudocódigo es la de poder expresar de una manera formal la implementación de una idea, de manera que cualquiera pueda llevarla a su lenguaje de programación preferido sin demasiado esfuerzo.

Un ejemplo podría ser:

```
# Obtener el máximo entero de una cadena
P = [5, 7, 1, 19, 31, 2]
d = -1
for p in P;
  if d > p ;
     d = p;
```

De este modo, podremos expresar soluciones genéricas a los problemas y que cada uno los lleve a su terreno favorito.

¿Dónde encontrar más recursos?

Todos los algoritmos que vamos a repasar, y **muchos más**, han sido estudiados, documentados e implementados decenas, cientos de veces.

Una sencilla búsqueda en internet nos llevará a multitud de artículos e implementaciones, y en la literatura existen multitud de referencias, siendo quizás la más conocida el famoso The art of Computer Programming, de Donald E. Knuth, aunque también destacaría *Introduction to Algorithms (3rd edition)* de Thomas H. Cormen.

Ordenación

El problema de ordenación de listas se podría plantear con el siguiente enunciado: *Dada una lista de entrada con* $N \geq 0$ *elementos comparables, devuelva una lista de salida con esos 'N' elementos en orden ascendente.*

Este problema es una gran introducción al análisis de algoritmos y de código fuente en general:

- Se suelen conocer de cursos de introducción a la programación, por lo que resultan familiares.
- La complejidad de los algoritmos va desde casos triviales hasta implementaciones recursivas.

Se presentan, a continuación, algunos de los algoritmos de ordenación más conocidos:

SelectionSort

La idea es sencilla: se busca el elemento más pequeño del array y se pone al principio. Se repite el proceso desde el elemento siguiente hasta que no se pueda iterar más.

Una sencilla implementación en Python podría ser la siguiente:

```
def selection_sort(lista: list) -> list:
    if len(lista) < 2:
        return lista

    for i in range(len(lista)):
        index_minimum = i
        for j in range(i+1, len(lista)):
            if lista[j] < lista[index_minimum]:
                index_minimum = j

        lista[i], lista[index_minimum] = lista[index_minimum], lista[i]

    return lista
```

Vamos a comentar los elementos nuevos del lenguaje que se han introducido, aunque lo mejor es, sin duda, que practiquéis vosotros mismos a implementar esto por vuestra cuenta:

1. Lo primero que puede chocar es la definición de la función.

   ¿No habíamos dicho que no hay tipos en Python? No exactamente, lo que no hay es tipado estático ni control en tiempo de ejecución.

   Para lo único que valen es para que las herramientas de *linting* o chequeo de código, que vienen integradas en muchos IDEs, nos den pistas de cuándo podemos estar haciendo algo mal durante la codificación de nuestro software.

   Esto es un proceso de **documentación intrínseca** que ayuda mucho al que revisa nuestro código a saber qué puede esperar del código que tiene delante.

2. Sentencia *range*: devuelve un elemento de tipo **iterable**, en este caso un **generador**, que nos devuelve de cada vez los enteros hasta el elemento anterior al último. Revisad la documentación enlazada para ver sus múltiples parámetros.
3. Intercambio de variables: la sentencia ***a, b = b, a*** en Python es capaz de intercambiar los valores de **a** y ***b*** sin necesidad de variables temporales.

Un equivalente en Java podría ser el siguiente:

```
public static int[] selectionSort(int lista[]) {
  if (lista.length < 2) {
    return lista;
  }

  for (int i=0; i<lista.length; i++) {
    int min_i = i;
    for (int j=i+1; j<lista.length; j++) {
      if (lista[j] < lista[min_i]) {
        min_i = j;
      }
    }
    int tmp = lista[i];
    lista[i] = lista[min_i];
    lista[min_i] = tmp;
  }
  return lista;
}
```

Estudio de complejidad:

En el caso de este algoritmo, **no existe** ni peor ni mejor caso: siempre se tiene que revisar toda la lista para cada elemento, por lo que se recorre siempre un número de veces proporcional al cuadrado del número de elementos.

$$n + (n-1) + \ldots + 1 = \sum_{i=0}^{n} i = n \cdot (n-1)/2 \Rightarrow O(n^2)$$

Insertion sort

Conocido como 'el método de la baraja', porque se asemeja a la manera en la que se mezclan las cartas de una baraja: se empieza por el i-ésimo elemento (*i > 0)* y se compara con los anteriores hasta que esté 'en su sitio'.

Expresado en pseudocódigo, el algoritmo podría ser algo así:

```
i = 1
while (i < A.size()) {
   j = i
   while (j > 0 and A[j-1] > A[j]) {
      intercambiar(A[j], A[j-1])
      j = j - 1
   }
   i = i + 1
}
```

Estudio de complejidad:

En el peor de los casos, vemos por el pseudocódigo que hay que recorrer el array una vez por cada elemento. Por tanto, como en el caso anterior, estamos ante una complejidad de $O(n^2)$.

Sin embargo, si el array de entrada está **totalmente ordenado,** la segunda condición del bucle más interno no se cumple jamás, ahorrándonos N-1 comparaciones de cada iteración y consiguiendo una ejecución lineal (*O(n)*) .

Lo que se puede inferir de esta propiedad es que este algoritmo funciona mucho mejor que el anterior cuando las entradas están parcialmente ordenadas. Cuanto más ordenadas estén, mayor será la velocidad de ordenación en promedio.

MergeSort

El algoritmo MergeSort es un algoritmo inventado en 1945 por John von Neumann que se basa en dividir la lista de entrada en sublistas, de forma recursiva, para después ordenar y juntar listas mucho más pequeñas.

Se introduce este algoritmo por dos razones fundamentales:

- Su complejidad temporal es **mucho mejor** que la de los anteriores, con un valor de *O(nlog(n))* en el caso promedio.

Para comparar, se muestran valores en una tabla que, si considerásemos los tiempos aproximados de ejecución según el tamaño de la entrada, nos muestran la ganancia significativa de la que estamos hablando:

n	**n^2**	**$n \cdot log(n)$**
10	100	23.025
100	10000	460.51
10^6	10^{12}	$13.38*10^6$

- Su solución implica entender el concepto de 'divide y vencerás', íntimamente relacionado con la **recursividad** que hemos visto en una sección anterior.

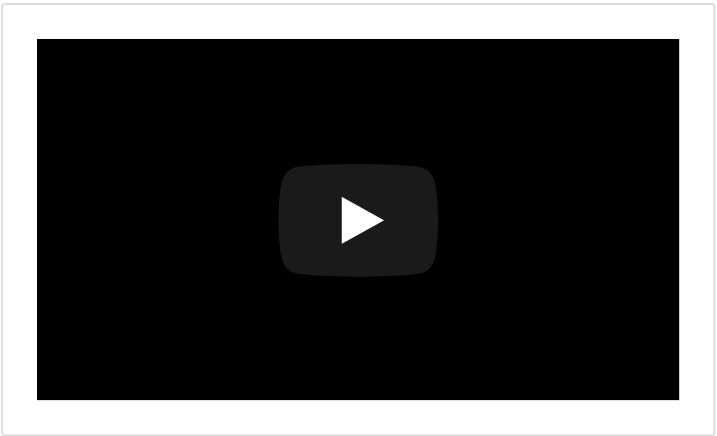
El algoritmo es el siguiente:

1. Se divide la lista de entrada en dos partes iguales.
    a. Si cada una de esas partes se puede **volver a dividir,** se hace.
    b. Si no se pueden dividir más, se **mezclan**: se juntan ambas listas en una con sus elementos ordenados.
2. El proceso termina cuando termina la recursión. Es decir, cuando se han mezclado todas las sublistas en una sola.

Es de destacar cómo se ha **dividido** el problema en partes más sencillas, que se resuelven todas de la misma manera. Es precisamente esta división en subproblemas lo que hace que la complejidad sea logarítmica en lugar de cuadrática.

Vídeo

Este vídeo de YouTube nos muestra, con gran acierto, una comparación entre diferentes algoritmos de ordenación. De una forma visual y efectiva vemos la gran diferencia en velocidad y consumo de memoria que existe entre ellos.



LA UNIVERSIDAD EN INTERNET | FORMACIÓN PROFESIONAL

PROEDUCA