Conceptos previos a Big Data

Competencias

- Conocerá los conceptos fundamentales y la importancia del análisis de algoritmos.
- Relación entre el Big Data y los algoritmos eficientes.
- Comprenderá el concepto de serialización en ciencias de la computación.
- Conocerá los distintos tipos de datos y su clasificación.
- Identificará el uso de las funciones map, filter, y reduce elementales de la programación funcional.

Motivación

Un punto de partida a considerar cuando comenzamos a estudiar Big Data es el hecho que muchas de las técnicas utilizadas han existido por años antes del advenimiento de esto. Conceptos como **Procesamiento Paralelo**, **Almacenamiento Distribuido** y **High-Performance Computing** por nombrar algunos se han desarrollado con anterioridad al concepto de Big Data.

Resulta que para entender los elementos que componen una solución Big Data, es necesario hablar de conceptos como **Complejidad Algorítmica, Formas de almacenamiento de datos y elementos de programación funcional**. A lo largo de esta lectura presentaremos los principales puntos a considerar cuando nos referimos a éstos.

Análisis de Algoritmos

La amplia cantidad de datos con los cuales nos enfrentamos como analistas de datos genera una serie de problemáticas. Una de ellas es cómo podemos hacer uso eficiente de nuestro código en tiempos de ejecución. Para entender cómo funciona esto, debemos manejar nociones básicas sobre **análisis algorítmico**. Mediante este lograremos entender qué le pedimos al sistema y cómo éste responde.

Informalmente, un algoritmo corresponde a una secuencia de pasos, que en un determinado orden y con uno o varios valores como entrada, generan uno o varios valores como resultado o salida. A su vez, y más relacionado al uso de algoritmos en computadores, las técnicas de análisis de algoritmos corresponden al estudio teórico del desempeño y utilización de recursos en los programas computacionales.

En la práctica, además de evaluar si un programa cumple con lo que se le pide hacer, es necesario evaluar otras múltiples dimensiones del código, tales como:

- Eficacia.
- Mantenibilidad.
- Confiabilidad.
- Modularidad.
- Simplicidad.
- Robustez.
- Replicabilidad
- Entre otros.

En esta unidad nos centraremos principalmente en la eficiencia y su importancia al momento de diseñar algoritmos a gran escala para el análisis de Big Data.

¿Qué relación tiene el Análisis de algoritmos con el Big Data y por qué los estudiamos?

Estudiar algoritmos nos ayuda a entender la **escalabilidad** de nuestras soluciones. Ya que cuando hablamos de Big Data, pensamos en aplicaciones que serán intensivas en el procesamiento de grandes cantidades de datos. Un código eficiente puede demorar horas en solucionar lo que un código mal construido puede demorar días. Esto no solo significa una tremenda pérdida de tiempo, sino también un pésimo uso de los recursos y, con ello, de dinero.

Por lo tanto, la construcción de soluciones de Big Data deben tener en consideración la **escalabilidad** como uno de los principales atributos de calidad. La confección de algoritmos eficientes corresponde a una habilidad con un gran sentido práctico. Computadores con grandes capacidades en hardware **NO** disminuyen el valor de los algoritmos veloces y eficientes. Es más, la gran mayoría de los cuellos de botella en los programas no se deben al hardware, sino más bien al software que no es eficiente en sus tareas.

El estudio de algoritmos nos permite comprender entre otras cosas:

- Es posible trazar una línea entre lo que es factible e imposible.
- El desempeño es la moneda de cambio en computación.
- Las matemáticas algorítmicas ofrecen un lenguaje para hablar acerca del comportamiento de los programas.

Eficiencia

La eficiencia de un algoritmo está asociada a *¿Qué tan óptimo es éste para resolver una tarea?* en comparación a otras alternativas. El desarrollo de un buen algoritmo buscará mantener tan bajo como sea posible el uso de recursos computacionales y el tiempo de ejecución en la máquina.

Para que un algoritmo sea práctico, debe organizar los datos de manera tal que apoye las tareas de procesamiento. Entonces, ¿ cómo podemos determinar cuándo un algoritmo es mejor que otro? Existe una estrategia teórica que está asociada al análisis matemático en el orden de complejidad de éstos. En la otra vereda está la alternativa empírica, que busca medir los tiempos de ejecución de distintos algoritmos para resolver una misma tarea. Si bien esta última suele generar resultados tangibles, tiene un contratiempo que hace un uso excesivo de los recursos disponibles en las máquinas.

Se debe tener en consideración:

- Se dice que un algoritmo es eficaz cuando logra el objetivo y resolución del problema planteado.
- Se dice que un algoritmo es eficiente cuando utiliza la menor cantidad de recursos posibles para lograr el objetivo planteado.

En ciencias de la computación, la eficiencia se analiza en función de tiempo y espacio:

- El espacio tiene relación con la cantidad de memoria utilizada por el programa.
- La eficiencia en tiempo corresponde a la medida necesaria en tiempo de ejecución de un programa.

En relación al cálculo del tiempo de un algoritmo, pueden incidir algunos factores propios de la máquina, como la capacidad del procesador. Para el cálculo matemático se debe abstraer de esos factores y concentrarse en el comportamiento del algoritmo.

Primer elemento a considerar: Datos de entrada

Para cada algoritmo, el comportamiento se ve afectado fuertemente por los datos que son proporcionados como entrada. Por ende, la eficiencia también se ve involucrada en función de la entrada. Como en la mayoría de los problemas existen infinitas posibles entradas, éstos no son analizados en función de cada instancia, sino en el tamaño de éstas. Es decir, el tamaño del objeto de entrada representado en un string, una matriz, el arreglo, la imagen, etc. Para algunos casos, se debe analizar las características de los datos de entrada. Por ejemplo, para los grafos es necesario analizar la cantidad de aristas y vértices del mismo.

Segundo elemento a considerar: Cálculo de eficiencia

No se busca calcular la eficiencia del computador en consideración de los múltiples factores que lo pueden afectar. Se busca calcular la eficiencia de los algoritmos. Es por esto, que en vez de medir la eficiencia en medidas de tiempo como segundos o microsegundos, la preocupación estará en medir la cantidad de veces que se debe ejecutar una operación primitiva. Es decir, una pregunta válida sería: ¿Cuántas sumas, multiplicaciones o validaciones booleanas son necesarias para ordenar un conjunto grande de datos?

Notación O (Big-O)

En general, lo relevante es tener una idea del número de operaciones que hará nuestro algoritmo y no un cálculo preciso para cada conjunto de datos. Es por eso que se suele utilizar la Notación O para describir el orden de magnitud del número máximo de operaciones que un algoritmo podría necesitar hacer. Este número dependerá principalmente del número de datos de entrada, que describiremos con la letra n.

Así, un algoritmo que, en el peor de los casos, deba realizar una operación por cada elemento de entrada tendrá un orden de $O(n)$, uno que deba hacer dos operaciones por elemento tendrá un orden $O(2 * n)$ y otro que compare cada elemento con todos los otros tendrá un orden $O(n^2 - n)$.

Debemos recordar que nos interesa analizar la eficiencia de nuestro algoritmo cuando se ejecuta sobre grandes volúmenes de datos, es decir, cuando el $n \to \infty$. A este tipo de análisis se le conoce como **Análisis Asintótico** y se basa en la idea que como el n será muy grande, siempre será posible encontrar una expresión que sirva como un límite superior al número de operaciones. Matemáticamente, buscamos que al dividir el número de operaciones por esta expresión, la división tienda a 0. De esta forma, sabremos que nuestra expresión sirve como un techo al número de operaciones que debe realizar nuestro algoritmo.

Principales variantes de la notación O

- **Complejidad algorítmica constante** $\rightarrow O(1)$**:** Se dice que un algoritmo es de tiempo constante si la operación primitiva a implementar no depende del tamaño del input de los datos. Un ejemplo en Python sería la siguiente función, la cual devolverá de manera irrestricta un 1, independiente del tamaño de los datos ingresados en ésta:

```
def constant_complexity(n):
   return 1
```

- **Complejidad algorítmica lineal** $\rightarrow O(n)$**:** Una complejidad algorítmica será de tiempo lineal si la operación primitiva a implementar afecta proporcionalmente el tiempo de ejecución. Un ejemplo en Python sería la siguiente función, donde añadimos un número al final de la lista `array` definida en la función. En la medida que n sea más grande, la función demorara más en implementar la expresión:

```
def linear_complexity(n):
   array = []

   for i in n:
       array.append(i)

   return len(array)
```

- **Complejidad algorítmica logarítmica** $\rightarrow O(log(n))$**:** Una complejidad algorítmica será de tiempo logarítmico si es que las operaciones primitivas implementadas dentro de la secuencia corresponden a operaciones de multiplicación o división. Por lo general esta complejidad se encuentra en árboles binarios, dado que el orden de operaciones relativo al tamaño del input de los datos tiende a disminuir en la medida que $n \rightarrow \infty$. Un ejemplo en Python:

```
def logarithmic_complexity(n):
    array = []

    while len(n) > 1:
        length_divided_by_two = len(n) // 2
        array = array + [n[0:length_divided_by_two]]
        n = n[0:length_divided_by_two]

    return len(array)
```

- Complejidad algorítmica de tiempo polinomial $\rightarrow O(n^d)$: Se dice que un algoritmo será de complejidad polinomial si es que el tiempo de ejecución de las operaciones primitivas corresponde a la cantidad de secuencias implementadas a lo largo del input de entrada. Por lo general, está determinada por la expresión más profunda dentro de los loops anidados. Un par de ejemplos en Python:

```
def quadratic_complexity(n):
    array = []

    for i in n:
        for j in n:
            array.append(j)

    return len(array)

def cubic_complexity(n):
    array = []

    for i in n:
        for j in n:
            for k in n:
                array.append(k)

    return len(array)
```

Para ejemplificar el comportamiento de cada uno de los casos de complejidad algorítmica, generamos la siguiente simulación. Evaluaremos el comportamiento en la cantidad de operaciones a realizar cuando ingresamos una lista con 100, 1000 y 2000 elementos. Para efectos prácticos, el código de replicación se encuentra en un archivo auxiliar `lec1_graphs.py`.

```
import lec1_graphs as afx
import matplotlib.pyplot as plt
plt.style.use('ggplot')
afx.compare_algo_complexities(evaluation_range=[100, 500, 1000, 2000],
                              log_values=True)
```

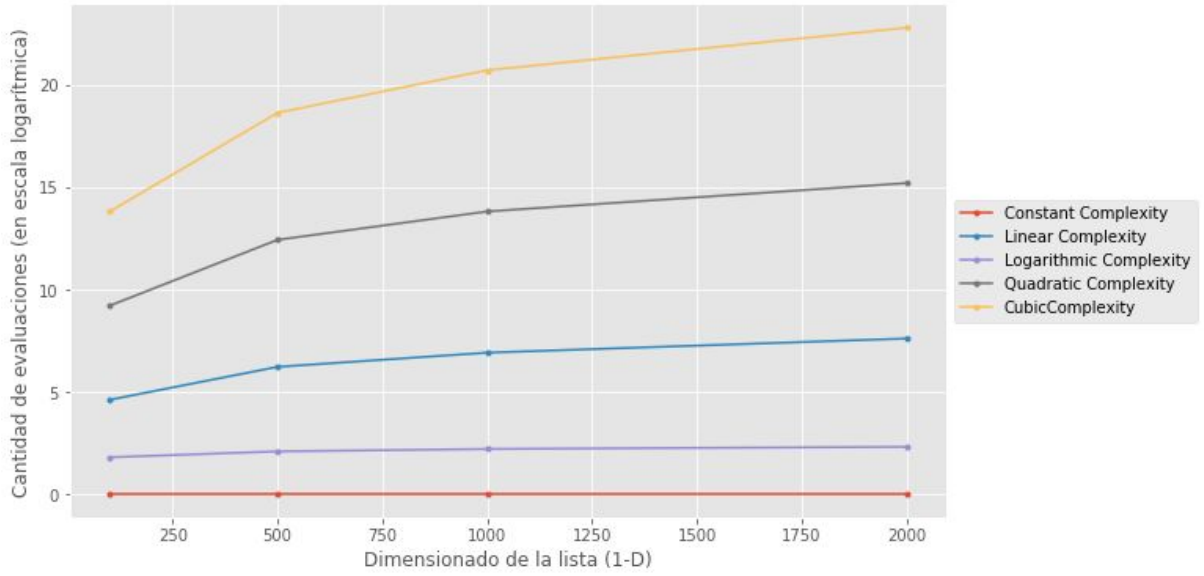

Imagen 1. Cantidad de evaluaciones vs Dimensionado de la lista (1-D).

Por motivos gráficos, el eje Y representa la cantidad de evaluaciones en la escala logarítmica. De esta forma logramos realizar comparaciones de una manera relativamente fácil. Consideremos el punto de partida, donde la cantidad de evaluaciones a realizar en el caso constante es de 1 (teniendo un logaritmo de cero). En el extremo opuesto, la complejidad cúbica cuando tenemos una lista de 100 elementos genera 1 millón de evaluaciones aproximadamente (el valor de la complejidad cúbica es de 13.8 en este escenario).

Un comportamiento a destacar a nivel general en este gráfico es el hecho que permite ejemplificar el orden de preferencia de la complejidad algorítmica. Obviando la complejidad constante, la segunda mejor opción es la complejidad de orden logarítmica, seguida por las complejidades lineales y polinomiales. Se infiere que en la medida que aumentamos la cantidad de términos polinomiales, aumentamos la cantidad de evaluaciones a realizar en una lista. Por último, por motivos de brevedad y sanidad mental, evitamos implementar un algoritmo de complejidad factorial $O(n!)$, considerado el caso menos óptimo.

Caso de estudio: Sorting Problem

Ya observando el comportamiento de los distintos algoritmos en la cantidad de operaciones primitivas a realizar, evaluemos el comportamiento de la complejidad algorítmica respecto al problema de ordenamiento. Muchos programas necesitan ordenar sus datos antes de ejecutar otras actividades. El ordenamiento es una tarea fundamental en ciencias de la computación, es por esto que existen distintas y variadas soluciones. ¿Cuál es el mejor? Principalmente depende de la cantidad de datos y la posición de estos antes de procesarlos.

A grandes rasgos el problema recibe una secuencia input con elementos $a1, a2, a3, \cdots, an$. Uno de los supuestos es que la posición de los elementos no refleja el valor contenido por cada uno de ellos. El objetivo es desarrollar un algoritmo que permita ordenar los valores del input en un retorno output que satisfaga $a'1, a'2, a'3, \cdots, a'n$ tal que: $a'1 \leq a'2 \leq a'3 \cdots \leq a'n$.

$$\texttt{input: [7, 3, 23, 4, 1, 9]} \rightarrow \texttt{output: [1, 3, 4, 7, 9, 23]}$$

En este experimento, compararemos el tiempo de ejecución entre dos variantes de sorting: Insertion y Bubble Sort.

Insertion Sort

Wikipedia nos indica que *"el ordenamiento por inserción es una manera natural de ordenar para un ser humano. Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay k elementos ordenados de menor a mayor, se toma el elemento k+1 y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se inserta el elemento k+1 debiendo desplazarse los demás elementos."*

Una implementación en Python de este algoritmo se vería así:

```
def insertion_sort(array):
    """
   dado una lista, implementar el algoritmo insertion sort
   """
    # para cada elemento en la lista
    for current_element in range(2, len(array)):
        # lo identificamos temporalmente
        key = array[current_element]
        # obtenemos su posición previa
        previous_element = current_element - 1
        # mientras este elemento previo sea mayor a cero
        # y su posición sea mayor a la identificación temporal
        while previous_element > 0 and array[previous_element] > key:
            # insertamos en el array el elemento previo en la posición
actual
            array[previous_element + 1] = array[previous_element]
            # el elemento previo retrocede otro espacio
            previous_element = previous_element - 1
        # actualizamos la posición
        array[previous_element + 1] = key
```

Bubble sort

También desde Wikipedia, *"el ordenamiento de burbuja funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiandolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada"*. El detalle de cómo implementar este algoritmo se escapa de esta lectura, pero se encuentra disponible en el artículo de Wikipedia.

La implementación de Bubble sort en Python sería más o menos:

```
def bubble_sort(arr):
    """
  dado una lista, implementar el algoritmo bubble sort
  """
    # extraemos la cantidad de registros en la lista
    n = len(arr)
    # para cada uno de los elementos alojados en la lista
    for i in range(n):
        # para todos los datos que preceden el elemento
        # Los últimos i datos ya se encuentran ordenados.
        for j in range(0, n-i-1):
            # si el elemento j es mayor al siguiente
            if arr[j] > arr[j+1] :
                # reordenar
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

```
import lec1_graphs as afx
import random; random.seed(5)
afx.compare_evaluations()
```

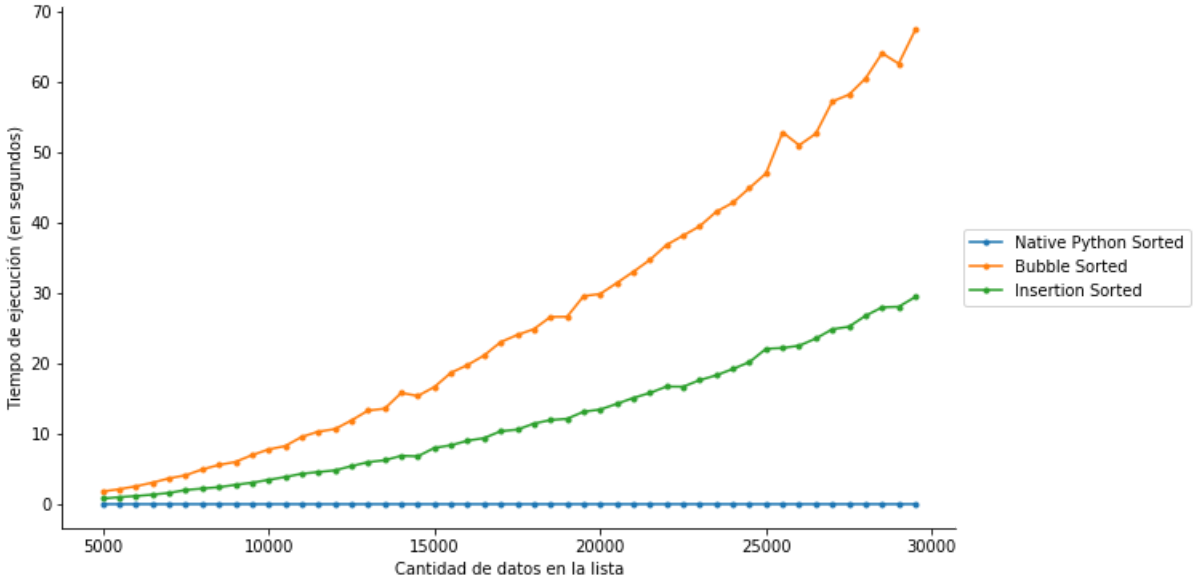

Imagen 2. Ejemplo Bubble Sort.

La función `afx.compare_evaluations` evalúa el tiempo de ejecución entre los algoritmos Insertion y Bubble en función a un rango de evaluación que refleja la cantidad de datos a ordenar en una lista. A grandes rasgos, observamos que el algoritmo Bubble Sort es mucho más lento que el algoritmo Insertion Sort. Esto se puede explicar dado que Insertion Sort es un algoritmo con una eficiencia entre $O(n)$ y $O(n^2)$, mientras que el Bubble Sort es un algoritmo que siempre tiene una eficiencia de $O(n^2)$.

¿Y por qué necesitamos saber sobre complejidad algorítmica en el contexto de Big Data?

Posteriormente cuando formalicemos nuestro conocimiento del paradigma MapReduce, aprenderemos sobre los pasos shuffling y sort implementados para ordenar las operaciones `map` antes de ser distribuídas a `reduce`. En este contexto, resulta que la elección del algoritmo en esta fase intermedia no es trivial, dado que se pueden producir bottlenecks: situación donde la aplicación se ve limitada por la velocidad de procesamiento.

En específico, el paso sorting en MapReduce ayuda al reducir a identificar cuándo una nueva tarea debería comenzar. Esto optimiza el tiempo de decisión del reducer en la entrega de resultados.

Datos: Volumen, Variedad y Velocidad

Hasta el momento tenemos conocimiento sobre el rol de los algoritmos en la entrega rápida y eficiente de resultados provenientes de datos. El otro elemento a tener en cuenta cuando hablamos de soluciones de Big Data son los datos y su naturaleza. Consideremos cuál era el escenario hace 30 años atrás. La mayoría de los datos que estaban disponibilizados a nivel de industria pertenecían a procesos bien definidos y mantenidos en bases de datos relacionales.

Hoy en día la situación ha cambiado drásticamente. No solo ha aumentado de manera exponencial la cantidad de datos, también han aumentado las fuentes originarias de éstos. Empresas privadas, organismos públicos, redes sociales, aplicaciones web y transacciones bancarias por nombrar algunas han impuesto nuevas formas de registrar, acumular y organizar los datos.

Todos los días se generan más de 2.5 quintillones de bytes y este número puede crecer rápidamente con la adopción del Internet of Things. Sólo en los últimos 2 años, se generó el 90% de todos los datos hasta ahora. Algunas cifras que ayudan a entender este crecimiento:

- Ahora, más de la mitad de las búsquedas web las realizamos a través de smartphones.
- Más de 3.7 billones de personas utilizan internet.
- En promedio, Google procesa más de 40.000 búsquedas cada segundo (3.5 billones por día).

Las redes sociales poseen un rol fundamental, a continuación algunas cifras de la actividad por minuto:

- Los usuarios de Snapchat comparten 527.760 fotos.
- Más de 120 profesionales crean su perfil en LinkedIn.
- Los usuarios ven 4.146.600 en YouTube.
- 456.000 tweets son enviados en Twitter.
- Los usuarios de Instagram postean 46.740 fotos.
- En Facebook, hay 510.000 comentarios publicados y 293.000 estados actualizado.

Categorías de datos

De manera adicional a la cantidad substancial de datos que se generan constantemente, hay que tener en consideración cómo se presentan estos. Si bien la mayoría de nuestros análisis siempre estarán orientados a generar matrices con filas y columnas, muchas veces nos enfrentaremos a datos cuya naturaleza carece de una estructura matricial por defecto, así como el tipo de elementos alojados dentro de ella. Los datos pueden ser organizados en tres categorías en función de su estructura:

Datos Estructurados

Partamos por el caso ubicuo: datos que están almacenados en filas y columnas. Este es el caso ideal, dado que tienen perfectamente definida la longitud en filas y columnas, los tipos de datos que estarán registrados en cada columna y las unidades de análisis definidas en las filas. Por lo general están definidos y almacenados en formato de tablas (hojas de cálculo y tablas de bases de datos relacionales) o archivos de texto plano con algún delimitador como comas, tabulaciones o espacios. En la actualidad, esta categoría de estructura de datos constituye aproximadamente el 10% de los datos totales disponibles. Por lo general reflejan un grado mayor de organización y la existencia de un equipo de analistas que están dedicados a definir su constitución. La siguiente imágen representa un ejemplo clásico de una base de datos estructurada.

	A	B	C	D	E
1	College Enrollment 2016 - 2017				
2	Student ID	Last Name	Initial	Age	Program
3	ST348-245	White	R.	21	Drafting
4	ST348-246	Wilson	P.	19	Science
5	ST348-247	Thompson	A.	18	Arts
6	ST348-248	Holt	R.	23	Science
7	ST348-249	Armstrong	J.	37	Drafting
8	ST348-250	Graham	S.	20	Arts
9	ST348-251	McFadden	H.	26	Business
10	ST348-252	Jones	S.	22	Nursing
11	ST348-253	Russell	W.	20	Nursing
12	ST348-254	Smith	L.	19	Business

Imagen 3. Base de datos estructurada.

No Estructurados

Cuando decimos que un conjunto de datos no son estructurados, hablamos de la ausencia de una serie de atributos que definen sus unidades de medición y qué atributos son medibles. Un aspecto importante a considerar con este tipo de datos es que dado que carecen de sentido per sé, es necesario definir mediante técnicas de extracción y re-dimensionalización su significado. Otro aspecto a considerar es que no presentan un formato específico. Pueden estar presentes como documentos de texto plano, documentos word, imágenes, audio, video.

La siguiente imágen representa un electrocardiograma.

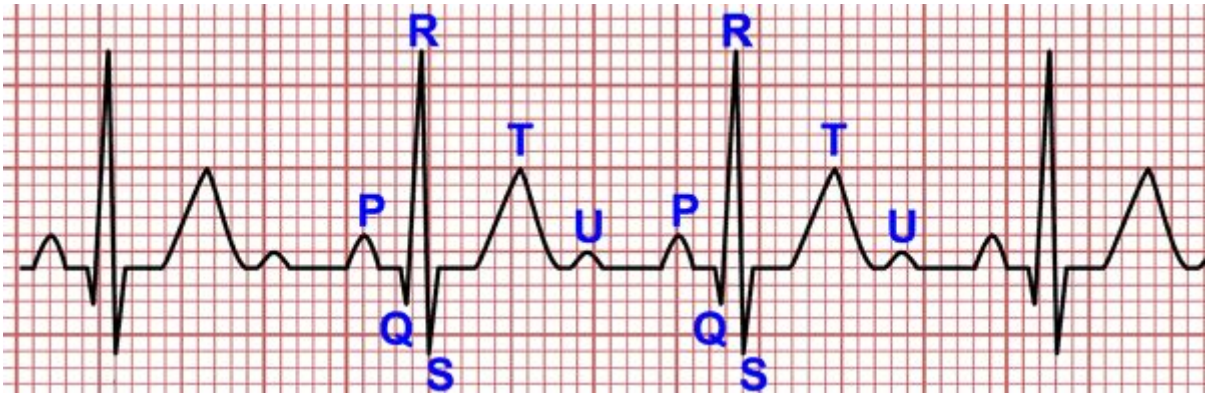

Imagen 4. Electrocardiograma.

Semi Estructurados

Por último, nos encontramos con los datos semiestructurados. Estos son una mezcla entre los estructurados y los no estructurados. Son una mezcla de los dos anteriores. No poseen una estructura perfectamente definida como los datos estructurados pero sí presentan una organización definida en sus metadatos donde describen los objetos y sus relaciones. En algunos casos están aceptados por convención; como por ejemplo los formatos HTML, XML o JSON.

Ejemplo:

```
{'id': 'ABW',
 'iso2Code': 'AW',
 'name': 'Aruba',
 'region': {'id': 'LCN',
  'iso2code': 'ZJ',
  'value': 'Latin America & Caribbean '},
 'adminregion': {'id': '', 'iso2code': '', 'value': ''},
 'incomeLevel': {'id': 'HIC', 'iso2code': 'XD', 'value': 'High income'},
 'lendingType': {'id': 'LNX', 'iso2code': 'XX', 'value': 'Not
classified'},
 'capitalCity': 'Oranjestad',
 'longitude': '-70.0167',
 'latitude': '12.5167'}
```

Principios de programación funcional

Una de los principios fundacionales de las soluciones en Big Data es el uso de operaciones en el paradigma conocido como MapReduce. Por ahora, quedémonos con la noción general que éste permite implementar operaciones de forma distribuida en un conjunto de datos mediante el procedimiento `Map` y el procedimiento `Reduce`.

En la actualidad, una de las tecnologías dominantes en la implementación de soluciones Big Data, Spark; se basa fuertemente en el paradigma de programación funcional mediante el lenguaje `Scala`. Mediante Spark, logramos abstraer el proceso de implementación de operaciones primitivas repetidas en un bucle, y reemplazar este proceso por la distribución de una operación a cada elemento existente en una estructura de datos definida. De esta manera, podemos evitar el problema de la ejecución **secuencial** de expresiones en los datos, evitando los cuellos de botella.

Para efectos prácticos del curso, trabajaremos con las operaciones `map`, `filter` y `reduce` de Python, así como con las comprensiones de lista. Estas operaciones tienen como objetivo el aplicar una serie de instrucciones o funciones a nivel de elemento.

De manera adicional, cabe destacar el hecho que hemos trabajado anteriormente con operaciones vectorizadas mediante las librerías `numpy` y `pandas`. Casos de operaciones como `numpy.logical_and`, `numpy.logical_or` y `numpy.where` son ejemplos de operaciones vectoriales enfocadas en la rapidez de los resultados.

Volvamos a evaluar la estructura de datos semi-estructurada definida anteriormente. Utilizando `requests.get`, podemos extraer los resultados en un formato `json` (JavaScript Oriented Notation).

```
import requests

# generamos un request de la api del banco mundial
datos_banco_mundial = requests\

.get("http://api.worldbank.org/v2/country?format=json")\
                        .json()
print("Dimensiones del json: ", len(datos_banco_mundial))
```

```
Dimensiones del json:  2
```

Si evaluamos la primera dimensión, esta hace referencia a los "meta datos" de request. Nos informa que existen un total de 304 registros y que accedimos a los primeros 50 en la página 1 de 7. Resulta que la segunda dimensión de nuestro json hace referencia a los datos, pero es una lista donde se encuentran los 50 elementos.

```
datos_banco_mundial
```

```
print("Tipo de objeto en la segunda dimensión: ",
type(datos_banco_mundial[1]))
print("Largo de la segunda dimensión: ", len(datos_banco_mundial[1]))
```

```
Tipo de objeto en la segunda dimensión:  <class 'list'>
Largo de la segunda dimensión:  50
```

Ahora accedemos al primer elemento. Sabemos que este corresponderá al registro de Aruba. Si revisamos su estructura, observaremos que este es un diccionario compuesto por algunas claves asociadas con valores únicos, pero otras llaves tendrán como valores a otros diccionarios.

```
aruba = datos_banco_mundial[1][0]
```

```
aruba
```

```
{'id': 'ABW',
 'iso2Code': 'AW',
 'name': 'Aruba',
 'region': {'id': 'LCN',
  'iso2code': 'ZJ',
  'value': 'Latin America & Caribbean '},
 'adminregion': {'id': '', 'iso2code': '', 'value': ''},
 'incomeLevel': {'id': 'HIC', 'iso2code': 'XD', 'value': 'High income'},
 'lendingType': {'id': 'LNX', 'iso2code': 'XX', 'value': 'Not
classified'},
 'capitalCity': 'Oranjestad',
 'longitude': '-70.0167',
 'latitude': '12.5167'}
```

Para ingresar a los valores asociados a las llaves, simplemente utilizando la notación `diccionario[llave]` lo lograremos. Si deseamos ingresar al valor asociado a una llave de un diccionario el cual está asociado a otra llave, podemos expandir el ejercicio a `diccionario[llave][llave]`.

```
print("Nombre: ", aruba['name'])
print("Capital: ", aruba['capitalCity'])
```

```
Nombre:  Aruba
Capital:  Oranjestad
```

```
print("Clasificación PIB: ", aruba['incomeLevel']['value'])
print("Región Geográfica: ", aruba['region']['value'])
```

```
Clasificación PIB:  High income
Región Geográfica:  Latin America & Caribbean
```

Nuestro primer "MapReduce"

Más adelante en el curso aprenderemos cómo implementar código de manera paralelizada con Hadoop y Spark. Pero por el momento, quedémonos con la implementación de un map y reduce primitivo para contar la cantidad de registros asociados a cada región. Podemos dividir la tarea en una serie de pasos:

Generar una función a mapear

Nuestra primera tarea es identificar cómo acceder a la región asociada a cada registro. Esto lo podemos lograr con la siguiente expresión:

```
aruba['region']['value']
```

```
'Latin America & Caribbean '
```

Ya identificando cómo acceder a un valor específico almacenado en nuestro `json`, podemos envolver esta expresión en una función y posteriormente pasarla mediante el método `map`.

```
def mapping_function(json_entry):
    return json_entry['region']['value']
```

Uno de los puntos a considerar es el hecho que esta función opera a nivel de una lista de jsons, la expresión a implementar debe estar contextualizada en función al elemento que aplicaremos el map.

```
# no está de más recordar que debemos envolver nuestro map en un list
region_ocurrence = list(map(mapping_function, datos_banco_mundial[1]))
```

```
# extraigamos las primeras 5 observaciones
region_ocurrence[:5]
```

```
['Latin America & Caribbean ',
 'South Asia',
 'Aggregates',
 'Sub-Saharan Africa ',
```

```
 'Europe & Central Asia']
```

Contar la cantidad de ocurrencias

Para contar la cantidad de ocurrencias de cada región debemos identificar las categorías únicas. Para ello podemos utilizar el método `set` que nos devolverá una colección no ordenada de elementos únicos.

```
unique_elements = set(region_ocurrence)
print(unique_elements)
```

```
{'South Asia', 'Aggregates', 'Sub-Saharan Africa ', 'East Asia &
Pacific', 'North America', 'Europe & Central Asia', 'Middle East & North
Africa', 'Latin America & Caribbean '}
```

En base a este objeto creado con `set`, estamos habilitados para contar la cantidad de ocurrencias de una palabra en la lista con `region_ocurrence.count('palabra')`. Si aplicamos esta expresión en una compresión de lista con `unique_elements` como un iterable:

```
[region_ocurrence.count(i) for i in unique_elements]
```

```
[3, 13, 6, 4, 2, 11, 2, 9]
```

Por último, podemos agregar el nombre de la región dentro de un diccionario:

```
[{i: region_ocurrence.count(i)} for i in unique_elements]
```

```
[{'South Asia': 3},
 {'Aggregates': 13},
 {'Sub-Saharan Africa ': 6},
 {'East Asia & Pacific': 4},
 {'North America': 2},
 {'Europe & Central Asia': 11},
 {'Middle East & North Africa': 2},
 {'Latin America & Caribbean ': 9}]
```

De manera alternativa, podemos implementar la clase `Counter` en la librería `collections` para llegar a un resultado similar.

```
from collections import Counter
Counter(region_ocurrence)
```

```
Counter({'Latin America & Caribbean ': 9,
         'South Asia': 3,
         'Aggregates': 13,
         'Sub-Saharan Africa ': 6,
         'Europe & Central Asia': 11,
         'Middle East & North Africa': 2,
         'East Asia & Pacific': 4,
         'North America': 2})
```

Transformando el json a un dataframe de pandas

Expandamos los ejercicios con funciones vectorizadas, ahora para convertir el `.json` a un dataframe de pandas. En este `DataFrame` vamos a preservar el nombre del país, su capital, la región geográfica, la longitud y latitud de la ciudad, la clasificación de ingreso y si ha solicitado fondos al banco.

El primer paso es desarrollar nuestra función a mapear. Un aspecto a considerar es que esta debe estar orientada a devolver una lista con todos los resultados

```
def json_to_row(json_entry):
    country = json_entry['name']
    capital = json_entry['capitalCity']
    region = json_entry['region']['value']
    longitude = json_entry['longitude']
    latitude = json_entry['latitude']
    income = json_entry['incomeLevel']['value']
    lending = json_entry['lendingType']['value']

    return [country, capital, region, longitude, latitude, income,
lending]
```

Tomemos por ejemplo el objeto `aruba` que contiene un `json` con los datos.

```
json_to_row(aruba)
```

```
['Aruba',
 'Oranjestad',
 'Latin America & Caribbean ',
 '-70.0167',
 '12.5167',
 'High income',
 'Not classified']
```

Con nuestra función testeada en un registro específico, ahora podemos mapearla en la lista de los datos. El resultado será:

```
import pandas as pd
datos_banco_mundial_df = pd.DataFrame(
    list(map(json_to_row, datos_banco_mundial[1]))
)
```

```
datos_banco_mundial_df.sample(5)
```

	0	1	2	3	4	5	6
36	Brunei Darussalam	Bandar Seri Begawan	East Asia & Pacific	114.946	4.94199	High income	Not classified
22	Bangladesh	Dhaka	South Asia	90.4113	23.7055	Lower middle income	IDA
30	Belize	Belmopan	Latin America & Caribbean	-88.7713	17.2534	Upper middle income	IBRD
24	IBRD countries classified as high income		Aggregates			Aggregates	Aggregates
6	Andean Region		Aggregates			Aggregates	Aggregates

Tabla 1. Datos Banco mundial.

Referencias

1. Cormen, T. (2001). Introduction to algorithms. Cambridge, Mass.: MIT Press.

2. Data Age 2025 (2017), IDC Analyze the Future.

3. Marr, B. (2018, July 09). How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read.

4. Skienna, Steve. The Algorithm Design Manual. Springer