# Trabajo Práctico 1
# 75.29 - Teoría de Algoritmos


**Alumnos:**

| Nombre | Padrón |
|---|---|
| Kevin Leon | 101441 |
| Nicolas Gatti | 93570 |
| Mariano Vazquez | 89059 |

# Índice

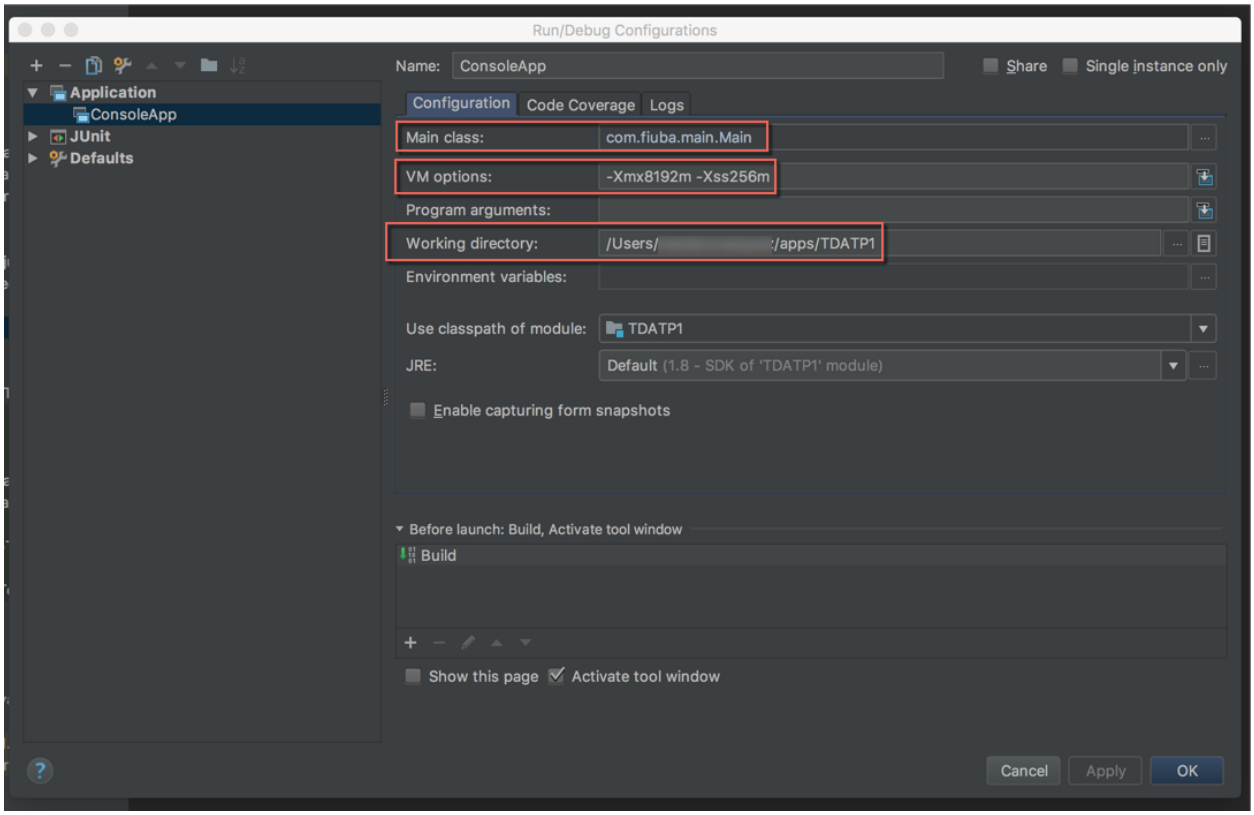
# Instrucciones para la ejecución del TP

El código de este Trabajo Práctico lo pueden encontrar en el siguiente repositorio de GitHub: https://github.com/gatti2602/TDATP1

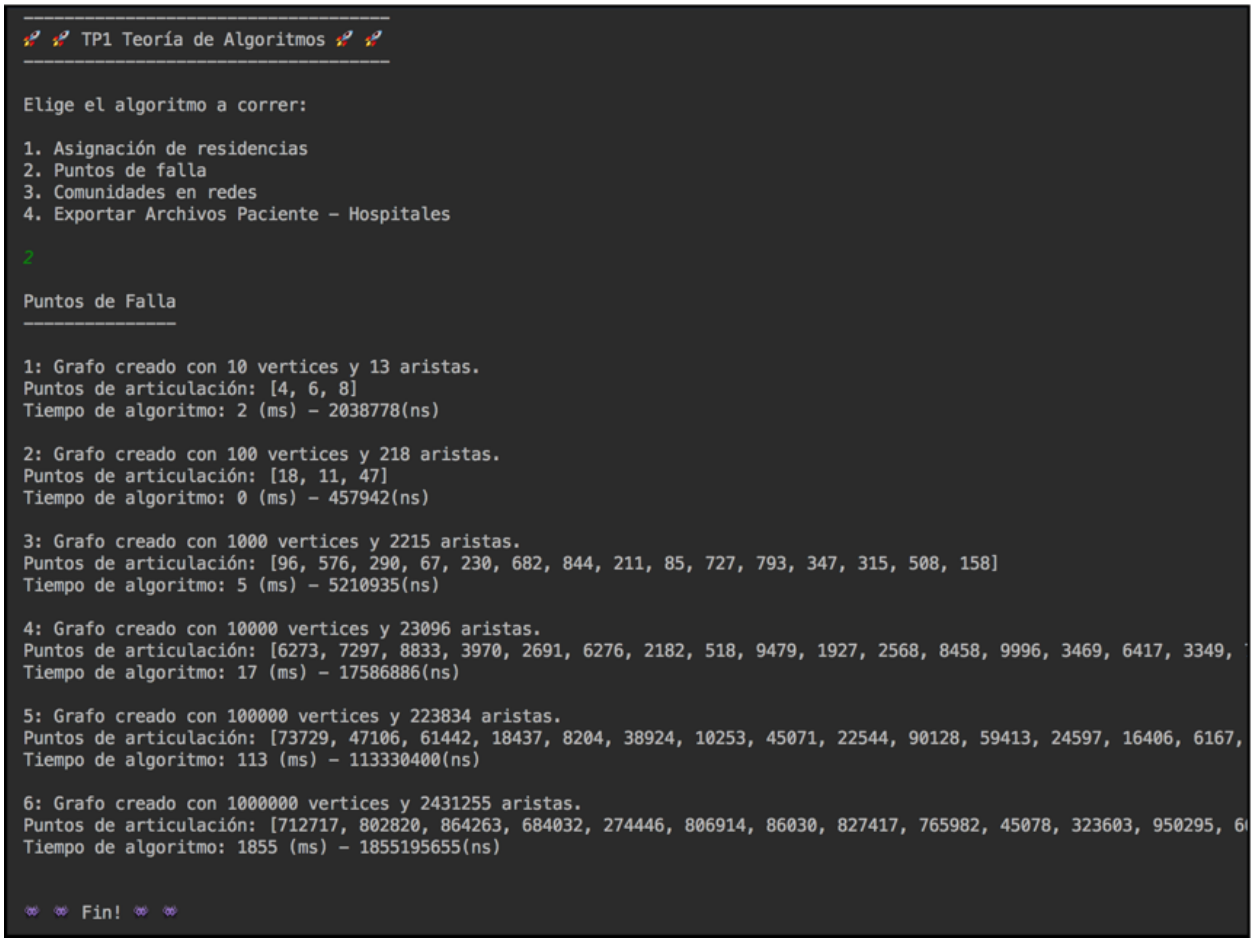
Comprende una librería donde está la implementación de los algoritmos, una aplicación de consola y tests. El código del programa ha sido escrito en Java.

Para la explicación de la ejecución de la aplicación de consola utilizaremos **IntelliJ**, pero es equivalente para cualquier IDE.

1. Crear una Run Configuration con la siguiente configuración:
    - Main class: **com.fiuba.main.Main**
    - VM options: **-Xmx8192m -Xss256m**
    - Working directory: *<path-donde-fue-descargado-el-código>*



2. Correr el programa para iniciar la aplicación. Elegir la opción deseada de acuerdo al algoritmo que se desee probar. Por ejemplo, si se desea probar la funcionalidad del punto **2. Puntos de falla**, se observará lo siguiente




**Nota:** hay ejecuciones que, debido a su gran cantidad de elementos, tardan mucho en terminar. Por ejemplo, para el punto 1, hemos tenido que esperar alrededor de 30 segundos para poder correr todas las configuraciones (n=m=1000, n=m=10000, n=m=100000).

# Asignación de residencias

## Reducción del problema simplificado

En base a las entradas aleatorias, el trabajo práctico nos pide reducir el **problema simplificado de la asignación genérica de residencias** al **problema de los matrimonios estables** descrito en la bibliografía. Para explicar la reducción, tenemos que plantear los parámetros de cada problema y encontrar su equivalencia. Si la encontramos, podemos utilizar el segundo problema para resolver el primero. Y si la equivalencia se realiza con un algoritmo de orden polinómico, dado que el problema de los matrimonios estables tiene también ese orden, podemos decir que la resolución del problema simplificado tendrá un orden de complejidad de orden polinómico.

### Problema simplificado de la asignación de residencias

```
n estudiantes
m hospitales públicos
q vacantes por hospital

cada estudiante --> listado de preferencias tamaño m (hospitales)
cada hospital   --> listado de preferencias tamaño n (estudiantes)
cada hospital   --> cantidad de vacantes (en un arreglo Q de tamaño m)
```

### Problema de los matrimonios estables

```
n estudiantes
n hospitales

cada estudiante --> listado de preferencias tamaño n
cada hospital   --> listado de preferencias tamaño n
cada hospital   --> cantidad de vacantes == 1 (en un arreglo Q de tamaño n)
```

Para realizar la equivalencia, debemos resolver estas diferencias:

1. **Tenemos varias vacantes por hospital.** Para resolver esto, podemos tratar a cada vacante *como un nuevo hospital*, agregando esta entrada a la *lista de hospitales* disponibles. Para los estudiantes, este nuevo hospital tendrá una preferencia similar al otro (se colocará contiguo en la lista de preferencias del estudiante).

2. **Tenemos más, igual o menor cantidad de estudiantes que de hospitales**.
    a. **(caso n > m) Puede haber más estudiantes que hospitales/vacantes disponibles**. Para resolver esto, podemos generar hospitales ficticios, con la menor preferencia posible, para igualar la cantidad de estudiantes y hospitales/vacantes.
    b. **(caso n < m) Puede haber más hospitales/vacantes disponibles que estudiantes**. Igual que en el caso anterior, podemos generar estudiantes ficticios, con la menor preferencia posible, para igualar la cantidad de elementos en cada colección.

El algoritmo de equivalencia se puede encontrar en la clase **com.fiuba.Reductor.java**

# Explicación del algoritmo

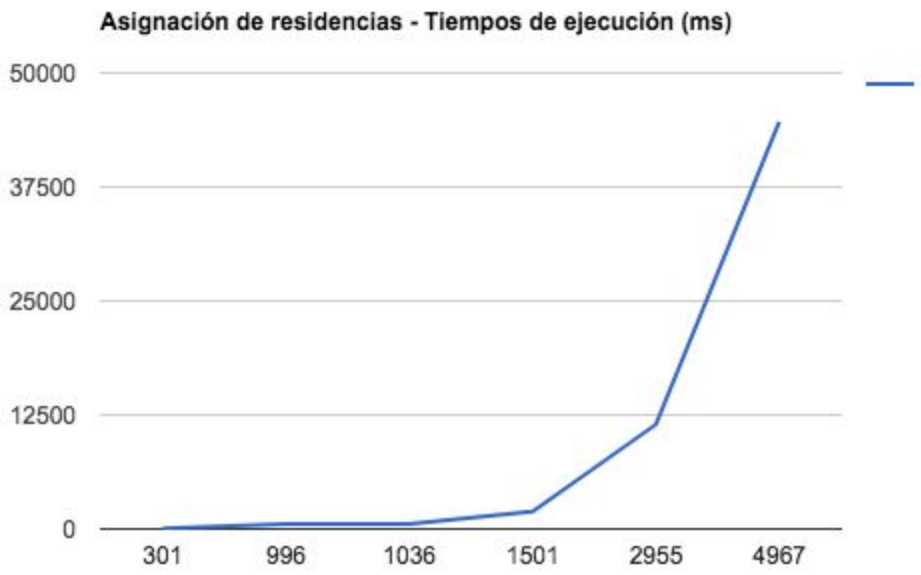
Se implementó el Algoritmo de Gale Shapley para encontrar parejas *estables* en base a dos set de elementos de igual tamaño. Este algoritmo garantiza que dicha solución estable puede ser encontrada con orden de complejidad en el peor de los casos de $O(n^2)$, siendo **n** el número de elementos de cada set de datos.

Además de ésto, se agregó el algoritmo que genera las equivalencias necesarias para la reducción, el cual también es de orden polinómico.

# Resultados de la simulación

Se realizaron diferentes corridas en base a la generación de instancias aleatorias. Dada la aleatoriedad de las vacantes, la cantidad de elementos que se le envía a Gale-Shapley no es determinística. Para extraer conclusiones, tomamos varias muestras entre 100 y 10000 y graficamos sus resultados.

| n | tiempo |
|---|---|
| 301 | 52 |
| 996 | 521 |
| 1036 | 570 |
| 1501 | 1920 |
| 2955 | 11469 |
| 4967 | 44658 |
| 4975 | 44971 |




## Conclusiones

Se puede observar que los resultados muestran que el tiempo de ejecución aumenta a medida que aumenta el **número de elementos** en los sets de datos. El orden de complejidad es polinomial, lo que implica que la reducción mantuvo ese orden, aunque no es $O(n^2)$ como Gale Shapley.

# Puntos de falla

## Explicación del algoritmo

Se implementó el Algoritmo de Hopcroft y Tarjan para encontrar puntos de articulación en grafos. Para ello, se implementó un algoritmo recursivo que recorrerá el grafo en base a una búsqueda en profundidad (DFS), de modo que navegará por cada nodo y sus hijos, regresando para el siguiente hermano luego de haber revisado todos sus hijos (backtracking).

Durante esta búsqueda, se almacenará un número de visita (o de profundidad) y un número bajo (*lownumber*) el cual será actualizado durante el backtracking con el valor más bajo de profundidad del nodo o sus descendientes.

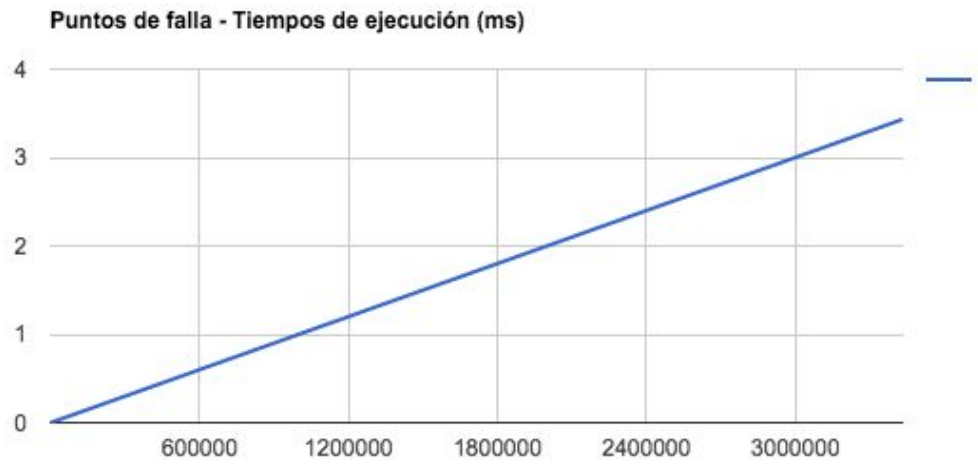
Luego del recorrido en profundidad, el vértice es un **punto de articulación** si

1. El número de visita es menor o igual al número bajo.
2. El vértice es la raiz (en nuestro caso, con ID igual a -1) y tiene más de 2 hijos.

## Resultados de la simulación

Se realizaron diferentes corridas en base a los archivos provistos por la cátedra. Estos son los resultados:

| | Archivo 1 | Archivo 2 | Archivo 3 | Archivo 4 | Archivo 5 | Archivo 6 |
|---|---|---|---|---|---|---|
| 1 | 1847725 | 509221 | 5551649 | 17493727 | 121764953 | 2599337759 |
| 2 | 1392861 | 546561 | 3569265 | 17281454 | 115580099 | 2118001153 |
| 3 | 1404095 | 608078 | 4865377 | 17597599 | 110495113 | 1978703294 |
| 4 | 1480826 | 566530 | 3826411 | 15368482 | 109513872 | 2539469910 |
| 5 | 1476243 | 474472 | 3889583 | 19669878 | 111646890 | 1947930782 |
| **Promedio (ns)** | **1520350** | **540972.4** | **4340457** | **17482228** | **113800185.4** | **2236688580** |
| **Promedio (ms)** | **1.52035** | **0.5409724** | **4.340457** | **17.482228** | **113.8001854** | **2236.68858** |
| **Promedio (seg)** | **0.00152035** | **0.0005409724** | **0.004340457** | **0.017482228** | **0.1138001854** | **2.23668858** |




## Conclusiones

Se puede observar que los resultados muestran que el tiempo de ejecución aumenta linealmente a medida que aumenta el número de **vertices** y **aristas** en el grafo. Esto, cumple con el orden de complejidad del algoritmo, el cual es $O(v + a)$.

# Comunidades en redes

## Explicación del algoritmo

El Algoritmo de Kosaraju, permite encontrar componentes fuertemente conexas en grafos dirigidos, manteniendo una eficiencia temporal $O(n+e)$ que en el peor de los casos es $O(n^2)$ con $n$ la cantidad de vértices.

La técnica que utiliza consiste en hacer una búsqueda DFS desde un origen cualquiera enumerando los vértices en el orden que se visitan. Una vez finalizado este proceso, se invierten los arcos del grafo y se vuelve a hacer DFS comenzando, y continuando al terminar la componente conexa, por el vértice de mayor peso. Las componentes conexas que resulten de este recorrido serán cada una de las componentes fuertemente conexas del grafo.

Esto sucede porque al invertir los arcos del grafo, estamos implicando que cualquier vértice alcanzable desde un vertice de peso alto, tuvo que haber sido alcanzado previamente (en el primer DFS) por el mismo, lo que hace concluir que esos dos vértices forman parte de una componente fuertemente conexa.

## Resultados de la simulación

Se realizaron diferentes corridas en base a los archivos provistos por la cátedra. Estos son los resultados:

| | Archivo 1 | Archivo 2 | Archivo 3 | Archivo 4 | Archivo 5 | Archivo 6 |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 5 | 25 | 231 | 5177 |
| 2 | 0 | 1 | 3 | 20 | 195 | 5327 |
| 3 | 0 | 1 | 4 | 23 | 197 | 5879 |
| 4 | 0 | 1 | 4 | 22 | 212 | 5104 |
| 5 | 0 | 1 | 3 | 21 | 187 | 4893 |
| **Promedio (ms)** | **0.2** | **1** | **3.8** | **22.2** | **204.4** | **5276** |
| **Promedio (seg)** | **0.0002** | **0.001** | **0.0038** | **0.0222** | **0.2044** | **5.276** |


# Conclusiones

En la implementación realizada en este trabajo se utilizaron listas de adyacencias para representar las distintas aristas de los grafos, lo que llevó a que el algoritmo se comporte con $O(n^2)$.

Esta decisión se tomó luego de, habiendo realizado la implementación con matrices de adyacencia, las simulaciones mas grandes dejarán a la máquina sin memoria heap y el proceso se interrumpiera. Considerando que se utilizó Java para el desarrollo, es posible que utilizando otro lenguaje que optimice el uso de memoria como C++ se pudiera implementar el algoritmo en un tiempo pseudo lineal $O(n+e)$.

# Anexo

- Enunciado del TP: https://algoritmos-rw.github.io/tda/tp1/#puntos-de-falla
- Repositorio del TP: https://github.com/gatti2602/TDATP1
- Tiempos de ejecución de algoritmos: https://docs.google.com/spreadsheets/d/1bWupD0h3mhTjg4H5NOP9L2HjRYyDLQ4NQwVOv02kazg/edit#gid=0
- Búsqueda en profundidad (DFS) https://es.wikipedia.org/wiki/B%C3%BAsqueda_en_profundidad