# 75.29 - Teoría de Algoritmos I: Trabajo Práctico n. 1

## Equipo Q:


Gomez, Juan (#87943)

Lavandeira, Lucas (#98042)

Rozanec, Matias (#97404)

Sbruzzi, Jose Ignacio (#xxxxx)

Facultad de Ingeniería, Universidad de Buenos Aires


9.abril.2018

# 1. Consigna

## 1.1. Lineamientos básicos

- El trabajo se realizará en grupos de cuatro personas (excepcionalmente de tres).
- La fecha de entrega es el lunes 9 de abril de 2018. Se debe entregar en el horario de clase en papel (informe + código en monoespacio), más una entrega en digital de código (.zip) e informe (.pdf) al correo de entregas del curso: tps.7529rw@gmail.com.
- El lenguaje de implementación es libre, pero se debe comunicar por correo en caso de no ser uno de: C, C++, Python, Java, JavaScript, Ruby, Go, Rust, Swift, Scala, Haskell, OCaml, Clojure, D, Lua, Elixir.
- Incluir en el informe los requisitos y procedimientos para su compilación y ejecución.

# Parte I
# Cálculo empírico de tiempos de ejecución

Implementar los siguientes algoritmos de ordenamiento para números enteros positivos:

- Selección
- Inserción
- Quicksort
- Heapsort
- Mergesort

a. Para cada uno de ellos analizar su complejidad teórica y compararlos (tiempo promedio y peor tiempo). Tener en cuenta las constantes para la comparación.

b. Construir 10 sets de números aleatorios con 10.000 números positivos.

c. Calcular los tiempos de ejecución de cada algoritmo utilizando los primeros: 50, 100, 500, 1000, 2000, 3000, 4000, 5000, 7500, 10000 números de cada set.

d. Estimar los tiempos medios de ejecución para cada rango-algoritmo y graficar.

e. Determinar para cada algoritmo anterior las características que debe tener un set para que se comporte de la peor forma posible (si el algoritmo lo permite).

f. Construir para cada algoritmo y para los rangos del punto "C" sets con las peores características y evaluar los tiempos de ejecución. Comparar con los generados con los sets aleatorios y graficar.

g. En base a los tiempos obtenidos compare con los valores teóricos y analice (Extensión máxima de 2 párrafos).

## Parte II

# Variante del algoritmo Gale-Shapley

Una liga amateur de Basketball tiene una manera extraña de iniciar la temporada. Un draft se realiza entre 200 jugadores anotados entre los 20 equipos que participaran. Tanto los jugadores como los equipos tienen una lista de preferencia donde establecen en orden decreciente sus elecciones. Cada listado es completo (tienen a todos los jugadores/equipos) y sin empates de preferencia. Se pretende construir un matching estable que termine con 20 equipos de 10 jugadores cada uno.

a. Construir el algoritmo de Gale-Shapley modificado para cumplir el requerimiento.

b. Probar que el mismo terminará en tiempo polinómico y siempre entregará un matching estable.

c. Ejecutar el algoritmo utilizando un set construido especialmente para el caso.

Información adicional:

- Cada equipo contará con un archivo llamado "equipo_[nro].prf" donde estarán en forma ordenada decreciente sus preferencias de jugadores.
- Cada jugador contará con un archivo llamado "jugador_[nro].prf" donde estarán en forma ordenada decreciente sus preferencias de equipos.
- Los jugadores estarán identificados por números entre el 1 y el 200.
- Los equipos estarán identificados por números entre el 1 y el 20.

## 2. Resolución

# Parte III
# Cálculo empírico de tiempos de ejecución

# Parte IV
# Variante del algoritmo Gale-Shapley

a. A continuación se presenta el pseudo código del algoritmo desarrollado.

**Data:** Prioridades de cada equipo y de cada jugador
**Result:** apareo óptimo entre equipos y jugadores
inicialización;
**while** *el equipo Ei tenga posiciones libres* **do**
  Ei le ofrece una posición a jugador Jk;
  **if** *Jk está libre* **then**
    acepta la posición;
  **else**
    /* Jk está en Ej */
    **if** *Jk prefiere a Ej* **then**
      se queda;
    **else**
      Jk se va a Ei
    **end**
  **end**
**end**

**Algorithm 1:** Pseudo código del algoritmo Gale-Shapley modificado.

A continuación se presenta el código implementado en Python.

```
#!coding=utf8

# Implementacion del algoritmo. Empecemos por la version 1:1...

from .queue import Queue


def gale_shapely(team_prefs: dict, player_prefs: dict) -> set:
    """

    :param team_prefs: { team: [player1, player2, ...] , ... }
```

```
    :param player_prefs: { player: [team1, team2, ...], ... }
    :return:
    """

    vacants = int(len(player_prefs) / int(len(team_prefs)))
    teams = {team: {'prefs': team_prefs[team], 'current': 0, 'vacants': vacants} \
            for team in team_prefs}

    teams_queue = Queue(*[team for team in teams])

    player_prefs = player_prefs.copy()
    for key in player_prefs:
        player_prefs[key] = {item: i for i, item in enumerate(player_prefs[key])}

    final_set = {}
    while teams_queue:
        team = teams_queue.pop()
        prefs = teams[team]['prefs']
        current = teams[team]['current']
        while teams[team]['vacants']:
            player = prefs[current]
            other_team = final_set.get(player)
            if other_team is None:
                final_set[player] = team
                teams[team]['vacants'] -= 1

            elif player_prefs[player][team] < player_prefs[player][other_team]:
                final_set[player] = team
                teams[team]['vacants'] -= 1

                teams[other_team]['vacants'] += 1
                if teams[other_team]['vacants'] == 1:
                    teams_queue.enqueue(other_team)

            current += 1
        teams[team]['current'] = current

    return set([(team, player) for player, team in final_set.items()])
```

**Termina en tiempo polinómico**
Para probar que el algoritmo propuesto terminará en tiempo polinómico, primero calculamos la cantidad máxima de iteraciones posibles:

$$\#max.\ de\ iteraciones = \#equipos * \#vacantes\ por\ equipo * \#jugadores$$

En nuestro caso:

$$20 * 10 * 200 = 40000$$

Esto significa que cada vacante de cada equipo le propone como máximo a todos los jugadores sin repetir.

**Función Variante**

$$\#max.\ de\ iteraciones - \#total\ de\ ofertas\ hasta\ el\ momento$$

Esta función es estrictamente decreciente porque las ofertas no se repiten (son únicas). Por lo tanto queda probado que el algoritmo termina.

**Matching perfecto**
El matching es perfecto si cada miembro de V aparece en un par de S. Como el algoritmo termina cuando no hay más vacantes disponibles, y además fue probado que el algoritmo termina, queda demostrado que no habrá elementos de V que no estén en S al finalizar el algoritmo.

De la misma forma, al haber igual cantidad de jugadores que de vacantes y dos vacantes distintas no pueden ser ocupadas por un mismo jugador, queda probado que no quedarán jugadores libres.

**Inestabilidades**
Supongamos inestabilidad $(e, p), (e', p') \in S/$

$$e \to [..., p', ...p, ...]$$

$$p' \to [..., e, ..., e', ...]$$

$(e, p) \in S \to$ e se postuló a $p$ y $p$ nunca lo echó.

¿Se pudo haber postulado $e$ a $p'$? Si no lo hizo, entonces contradice el ranking.

$e$ se postuló a $p'$ y $p'$ lo rechazó.

Llegamos a una contradicción. Queda entonces probada la estabilidad del algoritmo.