Trabajo Práctico 1

[75.29/95.06] Teoría de Algoritmos I
Primer cuatrimestre de 2018

Grupo MinMax

Alumno	Padrón	Mail
del Mazo, Federico	100029	delmazofederico@gmail.com
Djeordjian, Esteban Pedro	100701	edjeordjian@gmail.com
Kristal, Juan Ignacio	99779	kristaljuanignacio@gmail.com
Raveszani, Nicole	101031	nraveszani@gmail.com

Índice

1. Cálculo Empírico de Tiempos de Ejecución

1.1. Complejidad Teórica

1.1.1. Selección

De acuerdo a nuestra implementación[1] del ordenamiento de selección, la complejidad teóricas es:
$Tseleccion(n) = O(1+1+1+n*(1+n+(1+1+n(2)+1)+4)) = O(2n^2+9n+3)$

Esto viene de analizar linea por linea teniendo en cuenta:

- Hacer una asignación y comparar variables: $O(1)$
- Operadores aritméticos básicos: $O(1)$
- Sacar la longitud de un arreglo: $O(1)$
- Hacer un ciclo definido que itera por todo el arreglo: $O(n * f(n))$
- Hallar el indice de un elemento: $O(n)$

Finalmente, queda:
$PeorTselecccion(n) \in O(2n^2 + 9n + 3) \in O(n^2)$
$PromedioTselecccion(n) \in O(2n^2 + 9n + 3) \in O(n^2)$

1.1.2. Inserción

2) La sentencia de for en sí misma se considera de orden $O(1)$, pero lo importante es que a partir de esa sentencia se generarán n iteraciones (n = l) . Luego se tiene un orden de sentencia de $O(1 + nf(n))$, donde $f(n)$ representa la complejidad dentro del for.

3) Hacer una comparación y una resta tiene un orden de $O(2)$ (se asume que el return no se produce en la mayoría de los casos).

4) Hacer una suma y una comparación tiene un orden de $O(2)$. Como para todos los análisis se considera el peor caso, se debe considerara que esta comparación siempre da un resultado verdadero, y por ende, el algoritmo continúa (o bien, aunque fuera para la mayoría de los casos, con un n suficientemente grande, sería aproximadamente n).

5) Hacer una suma y asignarlo a una variable es $O(2)$.

6) La sentencia for en sí misma tiene un orden de $O(1)$, generando i iteraciones, con i la cantidad de elementos del sub array de los elementos predecesores al actual. Como n es arbitrariamente grande, entonces también debe ser así considerados el conjunto de elementos menores a uno dado (), pudiendo aproximar i n y por ende el orden termina siendo $O(1 + ih(n)) \approx O(1 + nh(n))$ con $h(n)$ dada por el análisis en los puntos siguientes.

7) Hacer una comparación tiene un orden de $O(1)$.

8) Asignar tres variables tiene un orden de $O(3)$.

9) Hacer un return tiene un orden de $O(1)$. Como se considera el peor caso, se asume que la mayoría de las veces el algoritmo continuará dando la condición anterior el valor de verdadero, y por ende, esta sentencia no se ejecuta.

[1] Las complejidades de los métodos nativos de las listas de Python 3 fueron sacadas de: https://wiki.python.org/moin/TimeComplexity

Finalmente se tiene un orden de: $O(1+1+1+n(3+2+2+1+n(1+3))) = O(3n^2+9n+3)$

Como es claro que encontrar en un arreglo un intervalo ordenado (suficientemente grande) es mucho menos probable que no hacerlo (es decir, para un arreglo de n elementos, de las $n!$posibles distribuciones de los elementos, sólo una estará ordenada, y con un n no demasiado grande, se cumple que $n! - 1 \approx n$). Luego, en el caso promedio se tiene que el algoritmo también tiene orden cuadrático.

Denominando al peor tiempo de ordenamiento por inserción como $PeorTinsercin(n)$: $PeorTinsercin(n) \in O(3n^2+9n+3)$

y análogamente, denominando al tiempo promedio del ordenamiento de inserción $PromedioTinsercin(n)$: $PromedioTinsercin(n) \in O(3n^2+9n+3)$

1.1.3. Quicksort

```
1  def quickSort(array):
2      if len(array) < 2 : return array
3      pivote = array[0]
4      menores, pivotes, mayores = [], [], []
5      for elemento in array:
6          if elemento < pivote:
7              menores.append(elemento)
8          else:
9              pivotes.append(elemento) if (elemento == pivote) else mayores.append(elemen
10     return quickSort(menores) + pivotes + quickSort(mayores)
```

1) Hallar la longitud de un arreglo es una operación $O(n)$, con n la cantidad de elementos del arreglo. A su vez, realizar una comparación es $O(1)$, y como para la mayoría de los casos se debe considerar que el return no se produce, se tiene una sentencia de orden $O(n)$.

2) Asignar una variable es $O(1)$.

3) Inicializar tres variables es $O(3)$.

4) La sentencia for por sí misma tiene un orden de $O(1)$, generando n iteraciones, con n la cantidad de elementos del arreglo. Luego se tiene un orden de sentencia de $O(1+nf(n))$, donde $f(n)$ representa la complejidad dentro del for.

5) Realizar una comparación es $O(1)$.

6) Agregar un elemento a una lista es $O(1)$.

7) La complejidad de la comparación ya está contemplada en el item 5.

8) Realizar una comparación es $O(1)$, y agregar un elemento a una lista también, por lo que se tiene una complejidad de $O(2)$, y por considerar el peor caso, es está la complejidad que se considera para la operación dentro del for (sumada a la de la primera comparación).

9) Para analizar la recursividad, se recurre al Teorema Maestro, ya que este algoritmo utiliza el principio de divisiòn y conquista. Sea tiene el tiempo de quicksort:

$Tquicksort(n) = a.Tquicksort(\frac{n}{b}) + f(n^d)$

La cantidad de llamadas recursivas (a) son dos. Para analizar b, se debe considerar dos casos. Por un lado, en el caso promedio, b será igual a 2, ya que en un arreglo arbitrariamente grande, es mucho más probable tomar en cualquier posición a un elemento alejado de las cotas superiores e inferiores que muy próximo a ellas, por lo que la recursión dividirá el problema en “dos

partes iguales" (probablemente nunca sean exactamente iguales, pero con un n arbitrariamente grande, las diferencias que puedan existir son despreciables en tanto el pivote elegido no esté lo suficientemente próximo a una cota superior o inferior). Pero existe un peor caso en donde el pivote elegido es una cota, y en general, siempre se toma una cota como pivote inicial (por ejemplo, si el arreglo ya estuviera ordenado, ya que en esta implementación siempre se toma como pivote al primer elemento del arreglo parametrizado). Para este peor caso (pero a su vez menos probable) se tiene que b=1, ya que todos (o casi todos, para n muy grande es indistinguible) los elementos quedan "a la derecha o a la izquierda" del pivote.

Por último, el tiempo de las llamadas no recursivas es $O(4n+4)$, por lo que $d = 1$ (siendo que la ecuación aproxima dicho orden al lineal). Se tiene entonces:

$PromedioTquicksort(n) = 2PromedioTquicksort(\frac{n}{2}) + O(n)$

$$PromedioTquicksort(n) = \begin{cases} n \log b & \text{si } a > b^d \\ n^{\log n} & \text{si } a = b^d \\ n^d & \text{si } a < b^d \end{cases}$$

Para este caso, como $2 = 2^1$, se tiene que $PromedioTquicksort(n) = (n \log n)$, por lo que:

$PromedioTquicksort(n) \in O(n \log n))$

Para el caso de b=1, no se puede aplicar el teorema del maestro, porque no hay "división y conquista" propiamente dicha (el arreglo no se subdivide apreciablemente). Para este caso, la recursividad funciona como un nuevo llamado a la estructura anteriormente analizada del código (items 1 a 8), por lo que en un peor caso donde el arreglo está ordenado, esta nueva llamada se produce n veces (nuevamente, por considerar n lo suficientemente grande), y para este caso se tiene una complejidad de $O(n(4n+4)) = O(4n^2 + 4n)$

Denominando al tiempo promedio de quicksort PeorTquicksort(n):

$PeorTquicksort(n) \in O(4n^2 + 4n)$

1.1.4. Heapsort

```
1  def heapSort(array):
2      heapify(array, len(array))
3      aux_heap = list(array)
4      array.clear()
5      while aux_heap:
6          array.append(heappop(aux_heap))
```

1) El heapify de un arreglo es $O(n)$ con n la cantidad de elementos del arreglo.

2) Duplicar el array es $O(n)$, y asignarlo a una variable es $O(1)$ por lo que para esta sentencia se tiene una complejidad de $O(n+1)$.

3) Vaciar un array es $O(1)$.

4) La propia sentencia del while tiene una complejidad de $O(1)$, y a su vez genera n iteraciones (ya que el heap tiene n elementos). Luego, la complejidad de esta sentencia es $O(1 + nf(n))$ con $f(n)$ la función que describe la complejidad del punto siguiente.

5) Eliminar un elemento del heap es $O(\log n)$ y agregarlo al array es $O(1)$. Luego se tiene que la complejidad de la sentencia es $O(1 + \log n)$.

$O(n + n + 1 + 1 + 1 + n(1 + \log n) = O(3n + 3 + n \log n)$

Denominando al peor tiempo heapsort como PeorTheapsort(n):
$PeorTheapsort(n) \in O(3n + 3 + n \log n$
y análogamente, denominando al tiempo promedio de heapsort PromedioTheapsort(n):
$PromedioTheapsort(n) \in O(3n + 3 + n \log n$

1.1.5. Mergesort

```
1  def mergeSort(array):
2          l = len(array)
3          if l < 2: return array
4          izq, der = mergeSort(array[:l//2]), mergeSort(array[l//2:])
5          resultado = []
6          while izq and der:
7                  if izq[0] < der[0]: resultado.append(izq.pop(0))
8                  else: resultado.append(der.pop(0))
9          resultado.extend(izq) if izq else resultado.extend(der)
10         return resultado
```

1) Hallar la longitud de un arreglo es una operación $O(n)$, con n la cantidad de elementos del arreglo. A su vez, asignar dicho valor a una variable es de orden $O(1)$, por lo que el orden total de esta sentencia es de $O(n + 1)$.
2) Hacer la comparación y el return tiene un orden de $O(2)$, pero como se asume el análisis de un caso con un n arbitrariamente grande, no se puede asumir que la mayoría de las veces se tendrá el return, por lo que la complejidad de esta sentencia pasa a ser $O(1)$.
3) Asignar dos variables sumado a dos operaciones da una complejidad de $O(4)$. La parte recursiva se analizará al final.
4) Asignar una variable es $O(1)$.
5) Hacer dos comparaciones es $O(2)$. A su vez esto genera n i iteraciones, ya que la suma de la parte izquierda como la derecha de todo el arreglo dan como resultado el arreglo entero. Luego la complejidad de esta sentencia es $O(2 + nf(n))$, donde $f(n)$ representa la complejidad dentro del while.
6) Realizar una comparación, un pop del primer elemento y un append son todas operaciones $O(1)$, por lo que la sentencia es $O(3)$.
7) Esta sentencia no realiza una comparación, por lo que considerando el peor caso, se asume que el algoritmo entra a la sentencia descrita en 6, y por ende, la 7 no se ejecuta.
8) Realizar una comparación es $O(1)$, y el extend agrega, en el peor de los casos, la mitad del arreglo original, por lo que la operación total es $O(1 + \frac{n}{2})$.
9) El return es $O(1)$.
Ni en el peor caso ni en el caso promedio se puede considerar que la lista tiene uno o ningún elemento, por lo que para analizar la complejidad de este algoritmo, considerando su recursividad, se recurre al Teorema Maestro. Se tiene el tiempo de mergesort (tanto en el peor caso como en el caso promedio):
$Tmergesort(n) = a.Tmergesort(\frac{n}{b}) + f(n^d)$
La cantidad de llamadas recursivas (a) son dos. Cada llamada recursiva divide la entrada inicial a la mitad (b=2). Por último, el tiempo de las llamadas no recursivas es

$O(n+1+1+4+1+2+n(1+1+1)+1+\frac{n}{2}+1) = O(\frac{9n}{2}+11)$ (asumiendo un n lo suficientemente grande como para que se requieran intercalar aproximadamente n elementos la mayoría de las veces), por lo que d=1. Se tiene entonces:

$Tmergesort(n) = 2.Tmergesort(\frac{n}{2}) + f(n)$ Por el Teorema Maestro: Como $2 = 2^1$, se tiene que $Tmergesort(n) = (n \log n)$

Luego, denominando al peor tiempo mergesort como PeorTmergesort(n): $PeorTmergesort(n) \in O(n \log n))$

y denominando al tiempo promedio de mergesort PromedioTmergesort(n): $PromedioTmergesort(n) \in O(n \log n))$

1.2. Comparación

1.2.1. Peores tiempos

$PeorTselecccin(n) \in O(3n2 + 9n + 2)$
$PeorTinsercin(n) \in O(4n2 + 9n + 2)$
$PeorTquicksort(n) \in O(4n^2 + 4n)$
$PeorTheapsort(n) \in O(3n + 3 + n \log n$
$PeorTmergesort(n) \in O(n \log n))$

Luego, el orden ascendente en eficiencia en el peor caso es inmediato: inserción, selección, quicksort, heapsort y mergesort.

1.2.2. Tiempos promedio

$PromedioTselecccin(n) \in O(3n2 + 9n + 2)$
$PromedioTinsercin(n) \in O(4n2 + 9n + 2)$
$PromedioTquicksort(n) \in O(n \log n))$
$PromedioTheapsort(n) \in O(3n + 3 + n \log n$
$PromedioTmergesort(n) \in O(n \log n))$

El orden parcial ascendente en eficiencia: inserción, selección y heapsort es inmediato. Para poder terminar de ubicar en la escala a quicksort y mergesort, se pone el foco en la complejidad de su parte no recursiva: por parte de quicksort es O(4n + 4), y en el caso de mergesort es $O(\frac{9n}{2} + 11)$. De aqui que el orden para tiempos promedios en orden ascendente es: inserción, selección, heapsort, mergesort y quicksort .

Es decir, la diferencia fundamental en ambos casos es que en el peor caso quicksort será mucho menos eficiente.

1.3. Tiempos de ejecución

Se presentan los tiempos de ejecución en segundos, redondeados a tres decimales, de cada algoritmo con cada intervalo de cada set.

1.4. Tiempos medios

Se calcularon los promedios de cada tiempo para cada cantidad de elementos y cada algoritmo:

1.5. Peores sets

1.5.1. Selección

Como se ve en el análisis, el comportamiento de este algoritmo no depende de las característicos del arreglo. En término más específicos, el algoritmo de selección no es un algoritmo "adaptativo": su conducta no se ve afectada por ninguna característica del array, por lo cual realizará el mismo número de comparación entre elementos tanto en el peor caso, en el caso promedio y en el mejor caso (considerando una longitud de arreglo arbitrariamente grande).

1.5.2. Inserción

Este algoritmo sí presenta un peor caso: cuando el arreglo de entrada se encuentra ordenado de forma descendente (y no al revés, según la implementación dada). El bucle interno del algoritmo deberá comparar cada elemento hasta llegar a la primera posición del conjunto, con lo cual realizará el máximo número de comparaciones posibles entre elementos. Esto también está contemplado dentro de su comportamiento en promedio: se destaca el caso en que, si el arreglo estuviera ordenado al revés, el algoritmo terminaría en un orden de O(n) (siendo para este caso trivial, mejor que cualquier ordenamiento).

1.5.3. Quicksort

Quicksort presenta un caso en que su comportamiento resulta cuadrático: cuando todos (o casi todos) los elementos son menores que el pivote seleccionado. En otras palabras, el pivote es una cota del set. En nuestra implementación, Quicksort elige siempre al primer elemento del arreglo como pivote, por lo que un set que cumpla con estas características es uno ordenado de forma descendente.

1.5.4. Heapsort

Para este algoritmo, el numero de comparaciones entre los elementos del arreglo puede variar en poca proporción dependiendo orden en que se presentan los mismos (según su ubicación en el heap). Aún así, tanto el caso promedio como el peor, su tiempo de ejecución será $O(nlogn)$, lo cual significa que asintóticamente no hay diferencia entre ambos, aunque pueden diferir en un valor constante.

1.5.5. Mergesort

Como se analizó con el Teorema Maestro, Mergesort es de orden $O(n \log n)$, por lo que no presenta un peor caso para ningún set en general.

1.6. Tiempos medios

Es claro que el caso de orden solo afecta descendente en quicksort e insertionsort. Se ve en la comparación que para pocos elementos el comportamiento de los ordenamientos es similar. Pasando los mil elementos se ve como quicksort e insertsort empeoran notablemente. Para el set mayor, se ve que insertionSort tarda cuatro veces más, y quicksort tarda más de 200 veces lo que tardaba en promedio. El resto de los ordenamientos no se ve afectado notablemente. En el

gráfico es más que notorio como quicksort tomó un orden cuadrático al igual que los ordenamientos por inserción y selección.

En el siguiente gráfico se muestra la comparación explícita del insertion sort: y a continuación el análogo de quicksort:

1.7. Comparación con valores teóricos

De acuerdo a las complejidades analizadas para el peor caso, se predice que los algoritmos de selección inserción y quicksort tardarán mucho más en ordenar arreglos que heapsort o mergesort. Esto se cumple, con la salvedad que en el análisis teórico para el peor caso se preveía que el ordenamiento por selección tendría peor desempeño que quicksort, lo cual no ocurrió, por ser el set especialmente diseñado para que quicksort se comporte de forma poco eficiente.

Con respecto al caso promedio, los tiempos se cumplieron de forma exactamente coherente con el orden de eficiencia predicho (en orden ascendente) en el análisis teórico: inserción, selección, heapsort, mergesort y quicksort. En síntesis, excepto por el peor caso de quicksort que no fue considerado de forma explícita, el análisis teórico se condice con los valores visualizados en la práctica.

2. Algoritmo Gale-Shapley

2.1. Algoritmo

El algoritmo de asignación propiamente dicho se reduce a la función homónima:

```
1 codigocodigocodigo
```

2.2. Demostraciones

2.2.1. Tiempo polinómico

Para facilitar la notación de está sección, J será la cantidad de jugadores y E la cantidad de equipos. Para la justificación detallada de cada sentencia del algoritmo, ver los comentarios de la complejidad en el código.

En el algoritmo el primer método que se llama es jugadoresEquiposGS. En caso en que se especifique la creación de los archivos .rtf, el método generarArchivos tiene una complejidad es de orden O(E*J), ya que por cada equipo se debe crear un archivo de longitud igual a la cantidad de jugadores, y lo análogo ocurre a la recíproca.

La carga de archivos con el método cargaDeArchivos naturalmente también de orden O(J*E) ya que por cada jugador que se quiera crear, se debe recorrer un archivo cuya longitud es igual a la cantidad de equipos, y viceversa.

La asignación, donde se genera el matching establece, tiene como condición de corte que no haya más vacantes en los equipos. Considerando esto, el hecho de que a continuación en el algoritmo se itere por cada equipo, y que se verifique para cada uno si todos tiene vacantes, es solo una especificación para lograr cubrir todas las vacantes, por lo que no agregan un orden diferente: simplemente es la forma de implementar que el algoritmo avance a partir de que se cubren distintas vacantes en distintas equipos. Esto implica que cada equipo deberá

completar todos sus lugares libres, y como hay igual cantidad de jugadores que de vacantes, el peor caso se considera cuando cada equipo debe preguntarle a todos los jugadores si desean ocupar una vacante: esto es O(J). Como hay E equipos, el orden de la asignación también es O(J*E).

Finalmente, almacenar el archivo de la asignación tiene un orden línea O(J+E). Esto se debe a que por cada equipo, se guarda su número seguido de los números de jugadores que tiene. La cantidad de jugadores que tiene un equipo es igual a sus vacantes (ver demostración en el próximo item de que cada equipo llena todas sus vacantes) y como la cantidad de vacantes totales es igual al número de jugadores, se tiene que se guardó un archivo con J + E números.

El orden cuadrático del algoritmo, entonces, es evidente, ya que es el peor caso de complejidad a lo largo de las funciones del algoritmo. Como O(J*E) es un orden polinomial, el algoritmo es de orden polinomial.

2.2.2. Matching estable

Para está demostración, se simplificará la estructura original de la solución implementada. Considerando el primer método llamado "jugadoresEquiposGS", se excluyen las características de la generación de los archivos .'prf', su carga, y el correspondiente almacenamiento del archivo donde se guarda la asignación estable. Entonces, el método se reduce a la asignación propiamente dicha.

Por una cuestión de comodidad, se puede traducir esto a un psuedcódigo basado en la documentación del método de asignación: "'Mientras haya un equipo con vacantes."' "'El equipo actual ofrece una vacante a su jugador favorito actual"' "'Si el jugador esta libre, acepta la vacante"' "'Si no esta libre, pero prefiere más al equipo actual, acepta la vacantey el otro equipo pierde a ese jugador."'

Como hay igual cantidad de jugadores que de vacantes totales, se ve que todos los equipos podrán cubrir todos su puestos. Supongamos que hubiera un equipo .ª"que ya ha .ºfrecido una vacante.ª todos los jugadores de acuerdo con su favoritismo, y ninguno la hubiera aceptado. Esto es decir que todos los jugadores ya han aceptado una vacante en un equipo previamente. Pero esto es una contradicción, porque existe la misma cantidad de jugadores que de vacantes totales, y por hipótesis hay una (al menos) vacante que .ª"no puede cubrir. Justamente, la contradicción viene de suponer que un equipo no ha podido satisfacer una de sus vacantes ofreciéndole a todos los jugadores, por lo que esto es falso. Con esto se ve además que la máxima cantidad de "proposiciones"que un equipo puede hacer es igual a la cantidad de vacantes disponibles en total, es decir, con la cantidad de jugadores.

Ahora se demostrar que una vez que cada equipo haya cubierto sus vacantes, no habrá parejas conflictivas (inestabilidades). La misma podría darse entre (j1,a) y (j2,b), donde j1 es un jugador que quiere estar en el equipo b, j2 es un jugador que quiere estar en el equipo a, a es un equipo que prefiere más a j2 sobre j1 y b es un equipo que prefiere más a j1 sobre j2.

Supongamos que a .ªlige primero". Si j1 aceptó la vacante en a, es porque j1 estaba libre, o bien, porque prefería estar en a que en otro equipo c (no necesariamente c = b).

Supongamos que era porque estaba solo. Luego b prefiere más a j2 que a j1, porque le ofreció su vacante antes. Si esto no fuera así, b le hubiera ofrecido antes su vacante a j1, y el mismo se hubiera cambiado de equipo, pero esto no

sucedió. En este caso se llega a un absurdo, porque por hipótesis b prefiere más a j1 (en otro caso la pareja no es conflictiva y no hay inestabilidad).

Entonces debemos suponer que j1 aceptó la vacante porque estaba en otro equipo c. Luego, se llega al mismo absurdo anterior, porque al fin y al cabo b termina con j2.

Luego, debe ser que b ."[el]igió primero". Si j2 aceptó la vacante en b, es porque j2 estaba libre, o bien, porque prefería estar en b que en otro equipo d (no necesariamente d = b o d = c). En cualquiera de los dos casos se llega a un absurdo similar al anterior. Todas los caminos a los que conduce la hipótesis "hay una inestabilidad en la asignación final"hacen llegar a una contradicción, por lo que queda probado que la asignación es estable.

No existe posibilidad que quede un equipo con alguna vacante libre, o un jugador sólo, como se probó al principio.

2.3. Ejecución

En el anexo junto con el código del algoritmo se ve la posibilidad que brinda de generar los archivos prf. El resultado de una de las ejecuciones fue la siguiente asignación:

```
1: 150 120 119 11 22 175 28 15 78 130
2: 182 200 8 191 44 36 60 134 17 159
3: 189 34 1 138 142 45 190 100 197 187
4: 88 9 132 46 62 91 188 103 145 70
5: 48 20 178 27 168 97 30 176 124 98
6: 125 129 25 86 180 165 106 5 177 12
7: 155 158 31 54 195 93 146 29 24 170
8: 164 63 110 127 109 140 96 160 149 143
9: 154 169 198 161 61 99 84 179 33 151
10: 173 69 35 117 163 65 112 118 104 43
11: 166 76 58 137 185 18 77 94 2 186
12: 51 4 171 199 87 162 19 181 85 80
13: 49 40 55 101 26 123 131 92 114 72
14: 193 115 39 37 135 6 172 38 141 174
15: 10 147 167 73 156 14 133 57 74 152
16: 126 192 136 148 71 23 52 153 111 32
17: 116 196 67 82 64 21 59 68 107 113
18: 105 56 90 66 47 183 7 75 108 16
19: 41 81 102 184 3 122 194 79 157 139
20: 42 83 13 128 53 50 144 89 121 95
```

A. Ejecución y compilación

A.1. Compilación

Como el trabajo fue realizado en python, no debe ser compilado. La versión de Python usada fue 3.6.1, en su versión de 64 bits (el código podría llegar a ser compatible con versiones previas de python 3, pero se recomienda utilizar la indicada).

A.2. Generación de scripts

El script generarSetRandom.py permite generar los sets aleatorios Por ejemplo, si se desea crear un set de 100 elementos con números del 1 al 100 con el nombre "setDePrueba.txt" se debe escribir al final del archivo:

generarSet("setDePrueba.txt", 1, 100, 100)

y ejecutarlo. O bien, importar el archivo y ejecutar la misma función:

```
>>> from generarSetRandom import generarSet
>>> generarSet("setDePrueba.txt", 1, 100, 100)
```

Para esto último, es necesario abrir python desde la ubicación donde se encuentra el archivo.

El generarSet es una "muestra aleatoria" de números que no se repiten, por lo que la cantidad dada en el tercer parámetro no debe superar el rango de las cotas. Es decir, no se puede pedir una cantidad de 100 de números del 1 al 10. Se eligió ese método para las pruebas para asegurar que siempre la muestra sea representativa.

La creación de los scripts ordenados de los peores casos se generaron también con este script:

```
>>> from generarSetRandom import generarSet
>>> generarSet("setDePrueba.txt", 1, 100, 100)
```

A.3. Pruebas de ordenamientos

Se pueden realizar las pruebas de cada ordenamiento ejecutando directamente el test correspondiente al ordenamiento que se desea probar. Todas las pruebas son iguales, e incluyen asserts con ordenamientos aleatorios. Ejemplo de ejecución: python heapTest.py

A.4. Medición de tiempos

Para esto se debe importar el archivo medicionTiempos.py, y llamar a la función medicionTiempos con la palabra clave de los archivos como parámetro. Por ejemplo, si los archivos se llaman "set" (set1, set2, etc.):

```
>>> from medicionTiempos import medicionTiempos
>>> medicionTiempos("set")
```

A.5. Stable matching

Para generar un matching estable, se debe importar el archivo matchinGS.py y ejecutar la siguiente línea: matchingGS('ruta de los archivos de jugadores y su correspondiente nombre sin número', 'extensión de los archivos jugadores', 'ruta de los archivos de equipos y su correspondiente nombre sin número', 'extensión de los archivos de equipos', 'nombre completo del archivo donde se guardará la asignación', cantidad de jugadores, cantidad de equipos, cantidad de vacantes por equipo, 'y' si se quieren generar los archivos o cualquier otro caracter en otro caso)

Por ejemplo, para cumplir con la consigna se requieren los siguientes comandos:

```
>>> from matchingGS import matchingGS
>>> matchingGS('archivos/jugador_', '.prf', 'archivos/equipo_', '.prf', 'asignacion.txt',
```

B. Tablas