

Universidad de Buenos Aires

Facultad de Ingeniería



75.29 Teoría de Algoritmos

Trabajo Práctico 1

Integrantes

- Andújar, Martín
- Moriello, Khalil Alejandro
- Gavrilov, Seva
- Guerini, Francisco Tomás

Primer cuatrimestre de 2018

Índice

1. Parte 1: Cálculo empírico de tiempos de ejecución	2
1.1. Consigna	2
1.2. Solución	2
1.2.1. A)	2
1.2.2. D)	7
1.2.3. E y F)	11
1.2.4. G)	13
2. Parte 2: Variante del algoritmo Gale-Shapley	14
2.1. Consigna	14
2.2. Solución	14
2.3. Correcciones	16
2.3.1. Modificación del algoritmo teórico	16
2.3.2. Implementación del algoritmo modificado	16
3. Instalación	18
3.1. Parte 1: Cálculo empírico de tiempos de ejecución	18
3.2. Parte 2: Variante del algoritmo Gale-Shapley	18

1. Parte 1: Cálculo empírico de tiempos de ejecución

1.1. Consigna

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).

1.2. Solución

1.2.1. A)

Algoritmo	Complejidad teórica(Promedio)	Mejor caso	Peor caso
Selección	$O(n^2)$	$O(n^2)$	$O(n^2)$
Inserción	$O(n^2)$	$O(n)$	$O(n^2)$
Quicksort	$O(n \log n)$	$O(n)$	$O(n^2)$
Heapsort	$O(n \log n)$	$O(n)$	$O(n \log n)$
Mergesort	$O(n \log n)$	$O(n)$	$O(n \log n)$

1.2.1.1. Demostraciones

1.2.1.1.1. Seleccíon

Dado nuestro código utilizado

```
def selection_sort(list_test) :
    for x in range(len(list_test) - 1, 1, -1):
        pos_of_max = 0
        for y in range(x, len(list_test)):
            if list_test[y] > list_test[pos_of_max]:
                pos_of_max = y
        swap(list_test, x, pos_of_max)
    return list_test
```

Algoritmo	Complejidad	Ejecuciones
for x in range(len(list_test) - 1, 1, -1):	C1	n
pos_of_max = 0	C2	n-1
for y in range(x, len(list_test)):	C3	
if list_test[y] > list_test[pos_of_max]:	[*]C3	n(n+1)/2
pos_of_max = y	C3	
swap(list_test, x, pos_of_max)	C4	(n-1) * 3 (porque son 3 lineas)
return list_test		

[*] Se llega al C3 porque se ejecuta de la siguiente manera: Primer loop : $2 \xrightarrow{\text{hasta}} n$ Segundo loop : $3 \rightarrow n-1$ Tercer loop : $4 \rightarrow n-2$ Ultimo loop : $n-1 \rightarrow 1$ Sumando todo da

$$n + (n-1) + +1 = n\left(\frac{n+1}{2}\right)$$

Tiempo Total

$$\begin{aligned}
 T(n) &= C1 \times n + C2 \times (n-1) + C3 \times \left(\frac{n(n+1)}{2}\right) + C4 \times (n-1) \times 3 \\
 &= C3 \times \frac{n^2}{2} + (C1 + C2 + \frac{C3}{2} + C4 \times 3) \times n + (-C2 - C4 \times 3) \\
 &= a^2 + b * n + c = O(n^2)
 \end{aligned}$$

1.2.1.1.2. Inserción

Dado nuestro código utilizado

```
def insertion_sort(list_test) :
    for y in range(1, len(list_test)) :
        key = list_test[y]
        x = y - 1
        while x > 0 and list_test[x] > key :
            list_test[x + 1] = list_test[x]
            x = x - 1
        list_test[x + 1] = key
    return list_test
```

Algoritmo	Complejidad	Ejecuciones
def insertion_sort(list_test) :		
for y in range(1, len(list_test)) :	C1	n
key = list_test[y]	C2	$n - 1$
x = y - 1	C3	$n - 1$
while x > 0 and list_test[x] > key :	C4	$[*] \sum_{x=1}^n t_j$
list_test[x + 1] = list_test[x]	C5	$\sum_{x=1}^n (t_j - 1)$
x = x - 1	C6	$\sum_{x=1}^n (t_j - 1)$
list_test[x + 1] = key	C7	$\$ n - 1 \$$
return list_test		

[*] Número de veces que se ejecuta el while para un valor de j

$$T(n) = C1n + C2(n - 1) + C3(n - 1) + C4 \sum_{j=1}^n (t_j) + C5 \sum_{j=1}^n (t_j - 1) + C6 \sum_{j=1}^n (t_j - 1) + C7(n - 1)$$

$$\sum_{j=1}^n (t_j - 1) = 1 + 2 + \dots + n - 1 + n = \frac{n(n - 1)}{2}$$

$$\sum_{x=1}^n (t_j) = \frac{n(n - 1)}{2} - 1$$

Sabiendo

Entonces :

$$= C1n + C2(n - 1) + C3(n - 1) + C4\left(\frac{n(n - 1)}{2} - 1\right) + C5\frac{n(n - 1)}{2} + C6\frac{n(n - 1)}{2} + C7(n - 1)$$

$$= \left(\frac{C4}{2} + \frac{C5}{2} + \frac{C6}{2}\right)n^2 + (C1 + C2 + C3 - \frac{C4}{2} - \frac{C5}{2} - \frac{C6}{2} + C7)n - (C2 + C3 + C4 + C5 + C6 + C7)$$

$$= an^2 + bn + c = O(n^2)$$

1.2.1.1.3. Quicksort

Implementación recursiva: Nota: en el trabajo se utilizó una implementación iterativa pero para explicar este punto se utiliza la implementación recursiva

```
def quick_sort_method(list_test, start, end) :
    if ( start < end ) :
        newIndex = partition(list_test, start, end)
        quick_sort_method(list_test, start, newIndex - 1)
        quick_sort_method(list_test, newIndex + 1, end)

def partition(list_test, start, end) :
    pivot = list_test[end]
    x = start - 1
    for y in range(start, end):
```

```

if list_test[y] <= pivot:
    x = x + 1
    swap(list_test, x, y)
swap(list_test, x + 1, end)
return x + 1

```

Algoritmo	Tiempo de ejecucion
def quick_sort_method(list_test, start, end) :	
if (start < end):	
newIndex = partition(list_test, start, end)	n
quick_sort_method(list_test, start, newIndex - 1)	i [tiempo en ejecutar primer subarray]
quick_sort_method(list_test, newIndex + 1, end)	n - 1 - i [tiempo en ejecutar último subarray]

Esto se traduce como

$$T(n) = cn + T(i) + T(n - 1 - i)$$

Para el caso promedio de todas las posibles particiones se escribe como:

$$\begin{aligned}
T(n) &= \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n - i - 1) + cn) \\
&= \frac{2}{n} [T(0) + T(1) + \dots + T(n - 2) + T(n - 1)] + cn
\end{aligned}$$

o bien,

$$\begin{aligned}
nT(n) &= 2(T(0) + T(1) + \dots + T(n - 2) + T(n - 1)) + n^2 \\
(n - 1)T(n - 1) &= 2(T(0) + T(1) + \dots + T(n - 2)) + c(n - 1)^2
\end{aligned}$$

-

$$\begin{aligned}
n(T(n)) - (n - 1)T(n - 1) &= 2T(n - 1) + 2cn - c \approx 2T(n - 1) + 2cn \\
nT(n) - (n - 1)T(n - 1) &= 2T(n - 1) + 2cn
\end{aligned}$$

o bien,

$$\frac{T(n)}{n + 1} = \frac{T(n - 1)}{n} + \frac{2c}{n + 1}$$

Aplicando un telescopio a $\frac{T(n)}{n+1}$:

$$\frac{2c}{n + 1} = \frac{T(n - 1)}{n}$$

y se obtiene la forma explícita:

$$\begin{aligned}
\frac{T(n)}{n + 1} + \frac{T(n - 1)}{n} + \frac{T(n - 2)}{n - 1} + \dots + \frac{T(2)}{3} + \frac{T(1)}{2} - \frac{T(n - 1)}{n} - \frac{T(n - 2)}{n - 1} - \dots - \frac{T(2)}{3} - \frac{T(1)}{2} - \frac{T(0)}{1} = \\
\frac{2c}{n + 1} + \frac{2c}{n} + \dots + \frac{2c}{3} + \frac{2c}{2}
\end{aligned}$$

Finalmente

$$\frac{T(n)}{n+1} = \frac{T(n)}{2} + 2c \sum_{j=3}^{n+1} \left(\frac{1}{j}\right)$$

Como n se hace muy grande, $\sum_{i=3}^{n+1} \left(\frac{1}{j}\right)$ se aproxima a $\ln(n) + \gamma$ donde γ es la constante de Euler $\approx 0,577\dots$

Por lo tanto

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c \ln(n) + 2c\gamma = \ln(n) + c2 = O(\ln(n))$$

Finalmente entonces

$$T(n) = O(n \log(n))$$

1.2.1.1.4. Heapsort

```
def heap_sort(list_test) :
    heap_size = len(list_test) - 1
    build_heap(list_test, heap_size)
    for x in range(len(list_test) - 1, 0, -1) :
        swap(list_test, 0, x)
        heap_size = heap_size - 1
        max_heapify(list_test, heap_size, 0)
```

Algoritmo	Tiempo de ejecución
def heap_sort(list_test) :	
heap_size = len(list_test) - 1	1
build_heap(list_test, heap_size)	n [por definicion]
for x in range(len(list_test) - 1, 0, -1)	n
swap(list_test, 0, x)	$3n$
heap_size = heap_size - 1	n
max_heapify(list_test, heap_size, 0)	$O(n \log n)$ [por definicion]

En promedio se realiza en n iteraciones un max_heapify que lleva $O(\log n)$, por lo tanto el costo temporal de este algoritmo en promedio es de $O(n \log n)$

1.2.1.1.5. Megesort

```
def merge_sort_method(list_test, start, end) :
    if start < end :
        middle = int(( start + end) / 2)
        merge_sort_method(list_test, start, middle)
        merge_sort_method(list_test, middle + 1, end)
        merge(list_test, start, middle , end)
```

Algoritmo	Tiempo de ejecución
def merge_sort_method(list_test, start, end) :	
if start < end :	n
middle = int((start + end) / 2)	n
merge_sort_method(list_test, start, middle)	$n/2$
merge_sort_method(list_test, middle + 1, end)	$n/2$
merge(list_test, start, middle , end)	n

El número de comparaciones de una lista con tamaño n satisface la forma recurrente:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + a(n)$$

$$1 \leq a(n) \leq n - 1$$

Resolviendo la ecuación de recurrencia usando $n = 2^m$ en la ecuación

$$T(n = 2^m) = 2T(2^{m-1}) + \alpha 2^m$$

se tiene

$$\begin{aligned}
T(2^m) &= 2T(2^{m-1}) + \alpha 2^m \xrightarrow{\times 2^0} T(2^m) - 2T(2^{m-1}) &&= \alpha 2^m \\
T(2^{m-1}) &= 2T(2^{m-2}) + \alpha 2^{m-1} \xrightarrow{\times 2^1} 2T(2^{m-1}) - 2^2T(2^{m-2}) &&= \alpha 2^m \\
T(2^{m-2}) &= 2T(2^{m-3}) + \alpha 2^{m-1} \xrightarrow{\times 2^2} 2^2T(2^{m-2}) - 2^3T(2^{m-3}) &&= \alpha 2^m \\
T(2^2) &= 2T(2^1) + \alpha 2^2 \xrightarrow{\times 2^{m-2}} 2^{m-2}T(2^2) - 2^{m-1}T(2^1) &&= \alpha 2^m \\
T(2^1) &= 2T(2^0) + \alpha 2^1 \xrightarrow{\times 2^{m-1}} 2^{m-1}T(2^1) - 2^mT(2^0) &&= \alpha 2^m
\end{aligned}$$

$$\boxed{T(1) = 0}$$

$$\boxed{T(2^m) = \alpha 2^m m}$$

Entonces, $\boxed{T(n) \approx \alpha n \log(n)}$

1.2.2. D)

1.2.2.1. Tiempo medio con 50 iteraciones:

Algoritmo	Tiempo medio (seg)
Heapsort	0.0003298004807970045
Selección	0.00013083834977933861
Mergesort	0.00023580199998107787
Quicksort	0.00012194083406313894
Inserción	0.00010067983583716966

1.2.2.2. Tiempo medio con 100 iteraciones:

Algoritmo	Tiempo medio (seg)
Heapsort	0.0007247297246397721
Selección	0.0004486266798213734
Mergesort	0.00047982986346184475
Quicksort	0.0002497603338049714
Inserción	0.0003789045405540037

1.2.2.3. Tiempo medio con 500 iteraciones:

Algoritmo	Tiempo medio (seg)
Heapsort	0.004888808857476334
Selección	0.011094667812789893
Mergesort	0.002757735193334554
Quicksort	0.001501715076367205
Inserción	0.01042211682185723

1.2.2.4. Tiempo medio con 1000 iteraciones:

Algoritmo	Tiempo medio (seg)
Heapsort	0.01082875558189933
Selección	0.04502248397095876
Mergesort	0.00584529233796971
Quicksort	0.003506901652045258
Inserción	0.0409498724160553

1.2.2.5. Tiempo medio con 2000 iteraciones:

Algoritmo	Tiempo medio (seg)
Heapsort	0.02383555138088056
Selección	0.17983277715210022
Mergesort	0.012822466433798141
Quicksort	0.007783576734286868
Inserción	0.16681784541239786

1.2.2.6. Tiempo medio con 3000 iteraciones:

Algoritmo	Tiempo medio (seg)
Heapsort	0.037735805689562696
Selección	0.4071431871011413
Mergesort	0.01896053938660458

Algoritmo	Tiempo medio (seg)
Quicksort	0.011680128970684933
Inserción	0.3761444555486815

1.2.2.7. Tiempo medio con 4000 iteraciones:

Algoritmo	Tiempo medio (seg)
Heapsort	0.05221940992985097
Selección	0.7219840037342387
Mergesort	0.026011349667937722
Quicksort	0.016142009466967977
Inserción	0.6706326836875611

1.2.2.8. Tiempo medio con 5000 iteraciones:

Algoritmo	Tiempo medio (seg)
Heapsort	0.06716955575572925
Selección	1.1285243604194548
Mergesort	0.032795204257913824
Quicksort	0.0222334473710184
Inserción	1.0522567008789494

1.2.2.9. Tiempo medio con 7500 iteraciones:

Algoritmo	Tiempo medio (seg)
Heapsort	0.10555483218916883
Selección	2.533056245873178
Mergesort	0.05055741457614715
Quicksort	0.03188779323241775
Inserción	2.3673165623728547

1.2.2.10. Tiempo medio con 10000 iteraciones:

Algoritmo	Tiempo medio (seg)
Heapsort	0.1456751018200908
Selección	4.51789061094763
Mergesort	0.06924800057395863
Quicksort	0.04411823011679061
Inserción	4.242444782562448

1.2.2.11. Gráfico para comparar todos los algoritmos con los tiempos medios

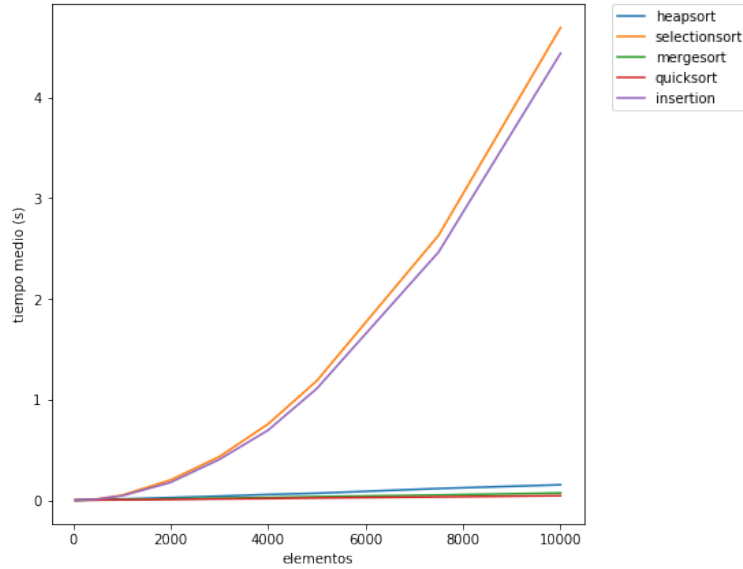


Figura 1: Tiempos medios

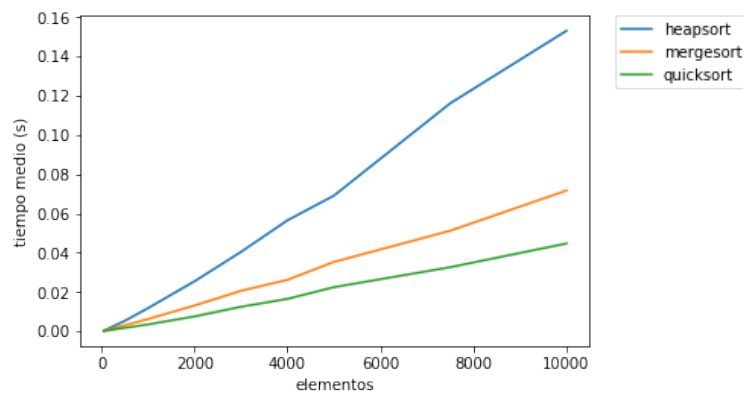


Figura 2: Tiempos medios sin considerar algoritmos cuadráticos

1.2.3. E y F)

Selección: En este algoritmo de ordenamiento no hay un set de datos que haga que se comporte de la peor manera posible ya que indistintamente de los datos el algoritmo hará la misma cantidad de comparaciones.

Inserción El peor rendimiento de este algoritmo se dará en casos en los que los datos estén ordenados de manera inversa a lo que se quiere obtener. Por ejemplo si se quiere ordenar de menor a mayor los enteros del uno al diez. Para obtener el peor caso los datos deberían estar ordenados de esta mayor a menor [10, 9, 8, 7, 6, 5, 4, 3, 2, 1].

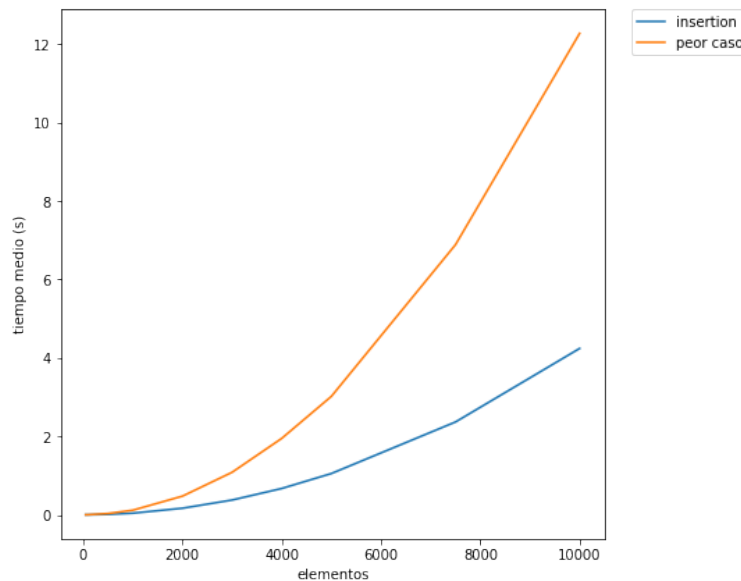


Figura 3: Inserción: Comparación con peor caso

Quicksort El algoritmo puede alcanzar el orden $O(n^2)$ en algunos casos, siendo este el peor escenario posible. El caso se da cuando los pivotes elegidos son los menores o mayores elementos del arreglo en todas las iteraciones recursivas del arreglo. Por ejemplo si se quiere ordenar el arreglo ordenado [1, 2, 3, 4, 5] de menor a mayor y si toma el último elemento del arreglo como pivote en cada iteración. Lo que resulta en que se haga la mayor cantidad de comparaciones, alcanzando el orden cuadrático.

Heapsort En el heapsort no importa el orden en el que se encuentren los datos ya que se realizan la misma cantidad de comparaciones (aproximadas) en todos los arreglos indistintos del orden o desorden de los datos. En cualquier caso el heapsort mantiene el orden $O(n \log n)$.

Mergesort En el mergesort la manera en el que se encuentran los datos ordenados no afecta el orden de operaciones que sigue siendo $O(n \log n)$ aun para el peor escenario posible.

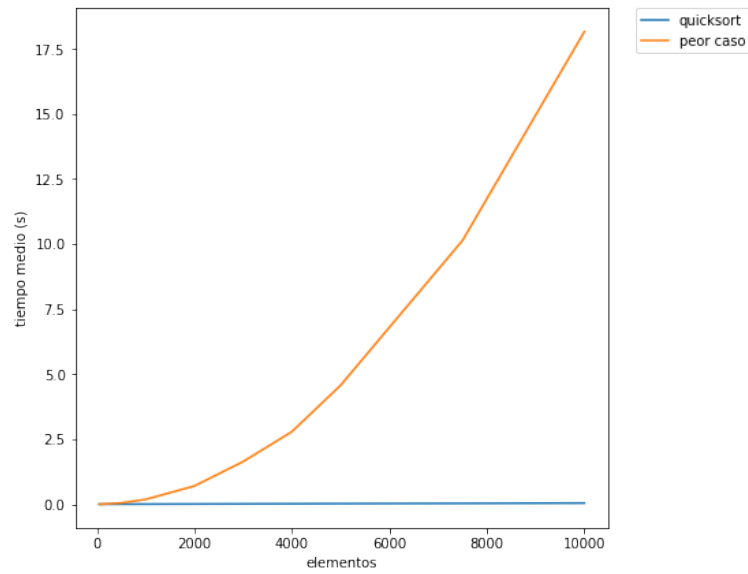


Figura 4: Quicksort: Comparación con peor caso

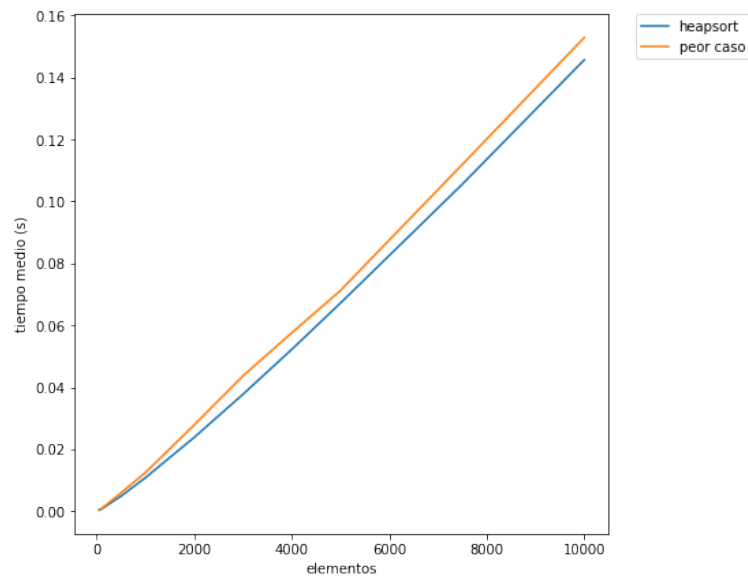


Figura 5: Heapsort: Comparación con peor caso

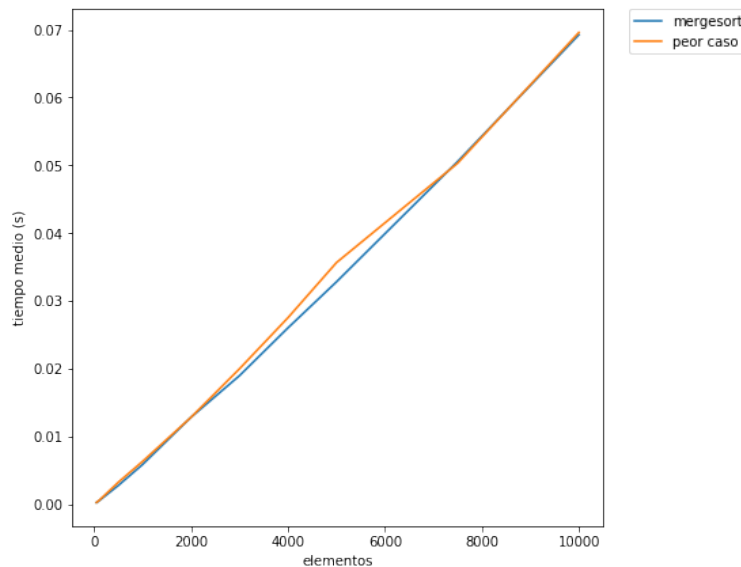


Figura 6: Mergesort: Comparación con peor caso

1.2.4. G)

- En los casos de SelectionSort e InsertionSort se puede ver que no hay demasiada diferencia con los tiempos medios de estos algoritmos con un sets de datos al azar, puesto que deben recorrer ciclos completos y hacer comparaciones, lo que no cambia mucho utilizar sets de datos considerados desfavorables.
- En el Quicksort la diferencia es abismal, al tener todo el array ya ordenado obliga a hacer todas las comparaciones y la diferencia se pronuncia aún más cuanto más grande es el tamaño del array
- Tanto en el HeapSort como en el MergeSort, la cantidad de comparaciones que se deben realizar son de $O(n \log n)$ por lo que casi no hay diferencia al compararse con arrays que puedan llevar a un peor caso.

2. Parte 2: Variante del algoritmo Gale-Shapley

2.1. Consigna

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.

2.2. Solución

Algoritmo de matching de Gale-Shapley devuelve un matching perfecto y estable entre dos conjuntos del mismo tamaño, y termina en tiempo polinómico ($O(n^2)$). El hecho de que sea perfecto nos asegura que no va a haber un elemento de cualquier conjunto sin matchear, y el hecho de que sea estable nos asegura de que en el matching resultante no haya ningún par, en el que un elemento prefiera una pareja de otro par, qué a su vez lo prefiere a él. El problema es que no es directamente aplicable, dado que la cantidad de jugadores y equipos no es la misma.

La solución que elegimos es extender los datos del problema para poder aplicarle el Gale-Shapley. Desde alto nivel, nuestra solución se puede explicar de la siguiente forma: en vez de 20 equipos, el segundo set se compone de 200 “vacantes”, de modo que para cada equipo hay 10 vacantes. Las preferencias de las vacantes son las mismas que las del equipo al que corresponden. En las preferencias de los jugadores los equipos se reemplazan por las vacantes, de manera que si antes un jugador prefería equipo 1 antes que equipo 2, ahora va a pasar a preferir todas las vacantes del equipo 1 antes que cualquier vacante del equipo 2.

Nuestra resolución del trabajo práctico entonces consiste en:

- 1) Convertir el set de entrada, cambiando el set de equipos por set de vacantes y las preferencias de los jugadores.
- 2) Aplicar Gale-Shapley
- 3) Convertir el matching resultante a la forma inicial, es decir, reemplazar las vacantes por el equipo al que corresponden.

La complejidad de cada uno de esos pasos es (siendo n = cantidad de jugadores = cantidad de vacantes):

- 1) Consiste en recorrer las preferencias de cada jugador y cambiarles las preferencias de equipos por vacantes. Eso es $O(n^2)$. Luego a cada vacante se le agregan las preferencias del equipo correspondiente, lo cual tampoco debería tomar más de n^2 iteraciones.
- 2) Gale-Shapley termina en $O(n^2)$ (por la demostración del libro).
- 3) Consiste en recorrer el matching una vez: $O(n)$

La complejidad total del algoritmo de esta forma va a ser de $O(n^2)$, osea que el tiempo es polinomial.

Observación: en nuestra implementación el ranking se calcula buscando un elemento en una lista de preferencias, la búsqueda en una lista de Python es $O(n)$, por lo cual, nuestra implementación de Gale-Shapley en realidad va a tomar n^3 iteraciones en el peor caso. Ese tiempo sigue siendo polinomial, pero es bastante facil convertir el cálculo de ranking en $O(1)$ usando una estructura de datos acorde.

Nos queda ver que el matching resultante es estable. Para eso queremos ver que no existan dos pares de jugador-equipo $(a, e1)$ y $(b, e2)$, donde el jugador a prefiera el equipo $e2$ y el equipo $e2$ lo prefiera al jugador a . Para obtener ese resultado después de la ejecución de nuestro algoritmo, el Gale-Shapley debió haber devuelto un matching con (a, u) y (b, w) , donde u es una vacante de $e1$ y w es una vacante de $e2$.

Supongamos que tal par existe. Eso significa en nuestra solución, que la última proposición de a fue hacía u . Si a no se le propuso a w previamente, entonces prefiere u por sobre w (osea que va a preferir cualquier vacante de $e1$ antes de cualquiera de $e2$), lo cual contradice la suposición de que a prefiere el equipo $e2$ por sobre $e1$. Si a se le propuso a w , entonces w rechazó su propuesta a favor de un jugador c (que puede ser el mismo que b , con el que terminó en el matching, u otro que prefiere por sobre b). Esto contradice la suposición de que $e2$ prefiere a por sobre b . Por tanto, tales pares no pueden existir y el matching es estable.

Otra solución posible e incluso mejor (más eficiente) a este trabajo práctico es modificando el algoritmo de Gale-Shapley para que se pueda matchear más de un elemento de un set al otro set (parecido a la solución del problema de hospitales y estudiantes). El orden de esa solución debería ser de $O(nm)$, donde n es la cantidad de jugadores y m es la cantidad de equipos.

2.3. Correcciones

Luego de la revisión del trabajo práctico se nos pidió reentregar, realizando una implementación más eficiente. Esto se debe a dos puntos principales:

- La premisa inicial con la que se encaró el trabajo práctico llevaba a una implementación poco eficiente. Esa premisa consistía en convertir los sets de entrada de $M \times N$ a $N \times N$, siendo N el set de mayor tamaño, para así solucionarlo como problema típico de matching estable en $O(n^2)$. Existe una solución en $O(nm)$.
- Para guardar las preferencias se utilizaba una lista, donde el índice de cada elemento indicaba el orden de preferencia. Eso llevaba a que cada vez que se comparaba el ranking de dos elementos, se realizaba la búsqueda de ese elemento en una lista ($O(n)$), convirtiendo la solución final en $O(n^3)$. Se puede utilizar otra estructura de datos para que el ranking se resuelva en $O(1)$.

2.3.1. Modificación del algoritmo teórico

Para conseguir que el matching se resuelva en $O(nm)$, donde n es la cantidad de jugadores y m es la cantidad de equipos, modificamos el algoritmo de Gale-Shapley de modo que cada equipo se pueda matchear con exactamente n/m jugadores. El algoritmo modificado se puede escribir en pseudocódigo de la siguiente forma:

Mientras algún equipo t_i sigue teniendo vacantes:

- t_i se le ofrece al jugador p_j , siguiente en su lista de preferencias
- si p_j está libre, se agrega al equipo t_i
- sino, significa que ya está en un equipo t_k
 - si p_j prefiere el equipo t_k , no hay cambios
 - sino p_j sale del equipo t_k e ingresa al equipo t_i

Como cada equipo se le ofrece a cada jugador como máximo una vez, y en cada iteración un equipo se le ofrece a un jugador, este algoritmo termina en $O(nm)$ iteraciones.

Para probar que el matching resultante es estable, suponemos que existen dos pares equipo-jugador (e_1, p_1) y (e_2, p_2) , donde jugador p_1 prefiere al equipo e_2 por sobre e_1 , y e_2 a su vez prefiere al jugador p_1 por sobre p_2 , formando así una inestabilidad.

Si p_1 no recibió ninguna propuesta de e_2 , todos los jugadores vinculados con e_2 tienen mayor ranking en las preferencias de e_2 , en particular p_2 , lo cual contradice que e_2 prefiere p_1 por sobre p_2 . Si p_1 recibió una propuesta de e_2 , significa que o estaba en otro equipo e_3 , al que prefiere por sobre e_2 , o que luego cambió a ese e_3 . Como en el matching final se encuentra en e_1 (que puede ser, o no, el mismo que e_3), significa que lo prefiere por sobre e_2 , llevando a una contradicción.

2.3.2. Implementación del algoritmo modificado

La implementación completa se puede ver en el archivo `tp1_2.py`, en el método `gale_shapley_fixed`. A continuación se explicará su funcionamiento y algunas mejoras para asegurar el orden $O(nm)$. Estas mejoras están ligadas a la implementación de las estructuras en Python (el lenguaje que utilizamos para resolver el trabajo práctico):

- Los equipos se le ofrecen a los jugadores.

- Las preferencias de los equipos se mantienen como una lista, por ejemplo: la lista de preferencias [1, 2] significa que el equipo prefiere al jugador 1 por sobre jugador 2. Para asegurarse que los equipos solo le hagan propuestas a jugadores nuevos (a los que no les hicieron propuesta todavía), antes de entrar al loop principal las preferencias se convierten en *iteradores*. *Iterador* es una abstracción de Python que permite iterar todos los elementos de una lista exactamente una vez y siempre conoce el siguiente elemento. Con su ayuda, podemos parar la iteración cuando un equipo tiene todas las vacantes ocupadas, y reanudarlo con el siguiente jugador en la lista de preferencias si un jugador acepta la propuesta de otro equipo.
- Como las preferencias de los jugadores no son recorridas en orden, sino son consultadas a medida que el algoritmo avanza, se utiliza un diccionario de Python para guardarlas, donde la clave es el equipo, y el valor es el orden de preferencia. De esta forma el ranking de un equipo para un jugador se resuelve en $O(1)$.
- El matching resultante se guarda en un diccionario, que tiene como claves a los equipos, y como valores a los jugadores vinculados con el equipo de la clave. Como necesito poder agregar y remover jugadores a lo largo de la ejecución del algoritmo, se los guarda en un *set*. La adición y la remoción de elementos en un *set* es $O(1)$.
- Para ver rápidamente si un jugador está vinculado con un equipo, y para saber qué equipo es, se utiliza un diccionario adicional de matching inverso, para que esas consultas se resuelvan en $O(1)$.

Complejidad de las estructuras de datos en Python: <https://wiki.python.org/moin/TimeComplexity>.

3. Instalación

Para este tp se usó python 3.5

3.1. Parte 1: Cálculo empírico de tiempos de ejecución

Para la primera parte es necesario instalar jupyter y matplotlib:

```
pip install jupyter matplotlib
```

Luego se debe ejecutar el siguiente comando para poder comenzar con las pruebas

```
jupyter notebook
```

Se debe dirigir hacia la carpeta tp1/tp1_1.ipynb y hay que ejecutar todas las celdas del notebook desde el principio hasta el final

Seleccionar consecutivamente en cada celda

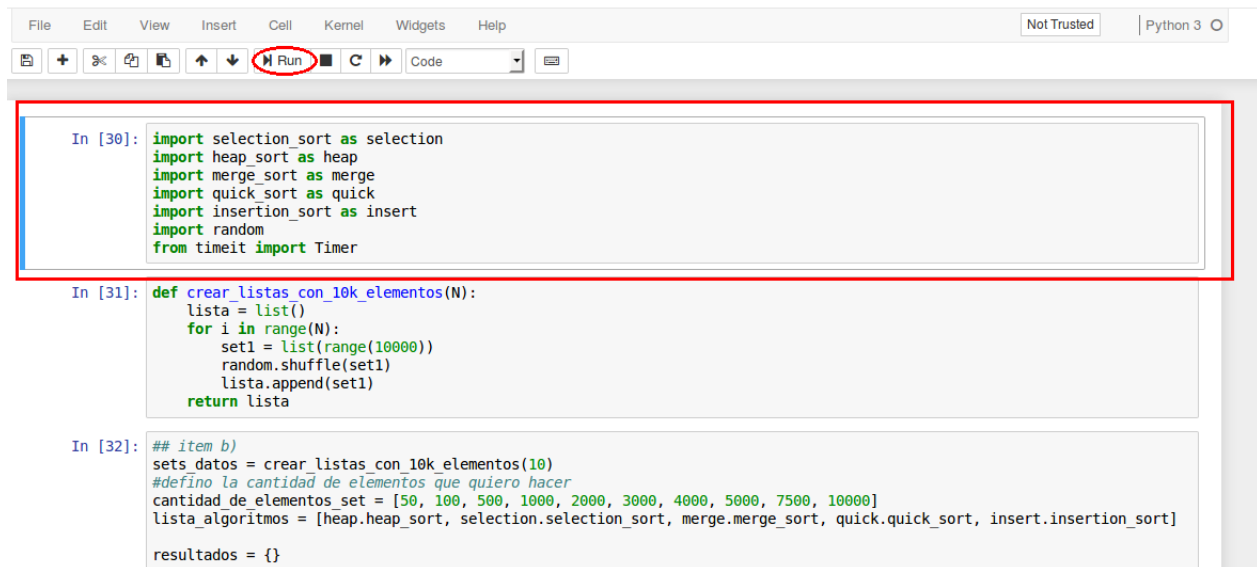


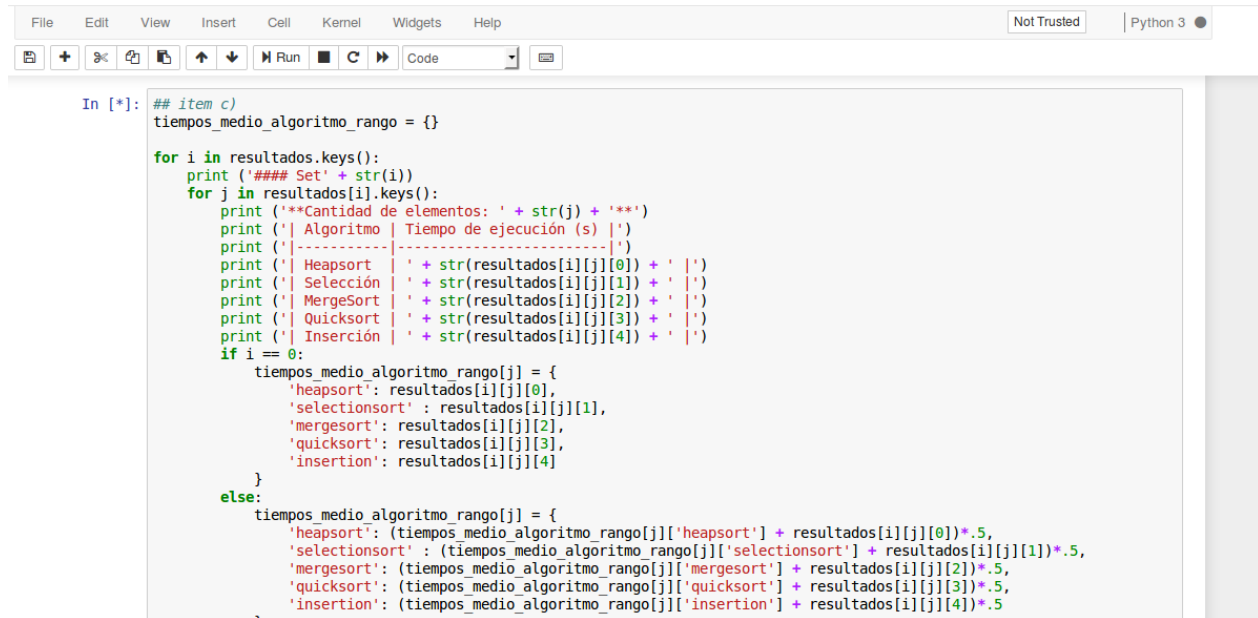
Figura 7

Cuando aparece un [*] significa que se debe esperar a que termine la ejecución para poder seguir ejecutando las otras celdas

3.2. Parte 2: Variante del algoritmo Gale-Shapley

Para la segunda parte, jupyter es opcional. Sirve para ejecutar el código interactivamente y ver la salida.

Para ejecutar el tp:



```

In [*]: ## item c)
tiempos_medio_algoritmo_rango = {}

for i in resultados.keys():
    print ('### Set ' + str(i))
    for j in resultados[i].keys():
        print ('**Cantidad de elementos: ' + str(j) + '**')
        print ('| Algoritmo | Tiempo de ejecución (s) |')
        print ('|-----|-----|')
        print ('| Heapsort | ' + str(resultados[i][j][0]) + ' |')
        print ('| Selección | ' + str(resultados[i][j][1]) + ' |')
        print ('| MergeSort | ' + str(resultados[i][j][2]) + ' |')
        print ('| Quicksort | ' + str(resultados[i][j][3]) + ' |')
        print ('| Inserción | ' + str(resultados[i][j][4]) + ' |')
    if i == 0:
        tiempos_medio_algoritmo_rango[j] = {
            'heapsort': resultados[i][j][0],
            'selectionsort': resultados[i][j][1],
            'mergesort': resultados[i][j][2],
            'quicksort': resultados[i][j][3],
            'insertion': resultados[i][j][4]
        }
    else:
        tiempos_medio_algoritmo_rango[j] = {
            'heapsort': (tiempos_medio_algoritmo_rango[j]['heapsort'] + resultados[i][j][0])*0.5,
            'selectionsort': (tiempos_medio_algoritmo_rango[j]['selectionsort'] + resultados[i][j][1])*0.5,
            'mergesort': (tiempos_medio_algoritmo_rango[j]['mergesort'] + resultados[i][j][2])*0.5,
            'quicksort': (tiempos_medio_algoritmo_rango[j]['quicksort'] + resultados[i][j][3])*0.5,
            'insertion': (tiempos_medio_algoritmo_rango[j]['insertion'] + resultados[i][j][4])*0.5
        }

```

Figura 8

`python tp1_2.py`

Por defecto, va a generar las preferencias randomizadas basadas en una semilla, va a crear una carpeta **preferencias** y guardar los archivos ahí. Los parametros por defecto se pueden alterar desde el código. Por ejemplo, para cambiar la semilla, la carpeta, y no generar archivos sino leer los que ya están, hay que modificar el código de la función main de la siguiente forma:

```

s = TPSolver(players=200, teams=20, seed=1000, path='mi_carpeta')
m = s.solve_tp_fixed(generate_files=False)

```

Para ejecutar la versión anterior a la reentrega, hay que modificar el método, al que se llama a `solve_tp`: `s.solve_tp()`.