Universidad de Buenos Aires

Facultad de Ingeniería

75.29 - Teoría de Algoritmos I

Cátedra Rosita Wachenchauzer

Trabajo Práctico No.1

79489 Rodriguez Leonardo Federico
84474 Paez Ezequiel Alejandro
87633 Rocha Mejía Maximo
99429 Ruiz Francisco

29 de abril de 2018

Índice

1. Parte 1: Cálculo empírico de tiempos de ejecución

Implementar los siguientes algoritmos de ordenamiento para números enteros positivos:

- Selección
- Inserción
- Quicksort
- Heapsort
- Mergesort

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

1.1.1. Selección

Del pseudocodigo podemos observar que existen dos bucles:

- bucle exterior
- bucle interior

Ambos son de ejecución obligatoria. El bucle exterior se ejecutará exactamente $n-1$ veces. El bucle interior también se ejecutará $n-1$ veces pero cada vez realizará una comparación menos que la anterior. Es decir, la primera vez se comparará el primer elemento a_1 contra los $n-1$ elementos restantes. La segunda vez a_2 contra los $n-2$ y así hasta llegar al elemento a_n que se comparará contra 1 solo elemento. Sumando todas las $n-1$ comparaciones del bucle interior se llega a la siguiente fórmula:

$$O((n-1)+(n-2)+...+1) = O(\sum_{i=1}^{n-1} i) \tag{1}$$

$$O(n^2) = \frac{n!}{2!(n-2)!} = \frac{1}{2}n(n-1) = \frac{1}{2}(n^2-n) \tag{2}$$

Este algoritmo requiere de la ejecución de todos los pasos independientemente del conjunto de datos motivo por el cual el tiempo promedio y el del peor caso es el mismo $O(n^2) = \frac{1}{2}(n^2-n)$

1.1.2. Inserción

Nuevamente el algoritmo realiza un recorrido de todo el array (bucle exterior for) realizando $n-1$ comparaciones. Para cada elemento lo compara contra todos los anteriores que ya se encuentran ordenados(bucle interior) es decir para el primer elemento no hay comparaciones, para el segundo elemento lo compara contra el primero y asi ... hasta llegar al n-ésimo elemento que debe ser comparado contra los $n-1$ elementos ya ordenados. Es decir, es el mismo orden que el algoritmo de selección con la salvedad que las comparaciones se realizan contra un array ordenado, con lo cual no hace falta realizar todas las comparaciones sino solo cuando el elemento de comparacion sea mayor que todos los restantes. De ahi que el bucle interior sea un while con lo cual en general es un poco más rápido que el algoritmo de selección con una salvedad. El algoritmo de selección solo realiza un swap por bucle interior, en cambio inserción realiza tantos swaps como elementos sean mayores que él en cada bucle interior. Esto es debido a la implementación elegida y en algunos casos como el nuestro puede ponderar más y terminar dando que el algoritmo de insercin tenga valores mayores que el de seleccion.

$$O(1+2+3+...+(n-2)+(n-1)) = O(\sum_{i=1}^{n-1} i)O(n^2) = \frac{1}{2}(n^2-n)$$

Esto es para el peor caso para el caso promedio el valor seria menor.

1.1.3. Quicksort

Quicksort es un algoritmo basado en la técnica de División y Conquista, estos algoritmos se caracterizan por ir dividiendo el problema en partes, resolverlo por separado para luego juntar las soluciones. En este caso, el algoritmo se llama a sí mismo 2 veces. Por lo tanto, al ser de División y Conquista, por el Teorema del Maestro tenemos que:

$$T_{QS}(n) = aT_{QS}(\frac{n}{b}) + O(n^c) \tag{3}$$

donde a representa la cantidad de llamadas recursivas que deben ser al menos 1, b es la cantidad de partes en las cuales se divide el problema que debe ser mayor a 1 y $O(n^c)$ el costo de trabajo hecho por fuera de las llamadas recursivas.

En el caso promedio, quicksort divide el vector en dos partes respecto de si son más grandes o más chicos que el número tomado como pivote. En estos casos, ambos vectores tienen una cantidad similar de elementos, haciendo que

el algoritmo sea más balanceado. Cuando sucede esto podemos decir que:

$$T_{QS}(n) = 2T_{QS}(\frac{n-1}{2}) + O(n) \tag{4}$$

Por el Teorema del Maestro, podemos concluir que si $log_b(a) = c$ entonces este algoritmo tiene como cota a $O(nlogn)$, estando muy cerca al mejor caso.

Ahora veamos el peor caso, este sucede cuando uno de los vectores en los que se divide el problema queda vacio y el otro tiene los n-1 elementos restantes. Esto sucede porque el pivote elegido es unos de los extremos de la lista.

Por lo tanto, a cada llamada recursiva, se va a ir ordenando de a un elemento por vez, haciendo que se perezca mucho en su rendimiento a los ordenamientos de selección y de inserción. Su análisis computacional nos lleva a:

$$T_{QS}(n) = T_{QS}(n-1) + O(n) = \sum_{k=1}^{n} O(k) = O(n^2) \tag{5}$$

Para concluir, podemos ver que es algoritmo que más diferencia tiene entre el caso promedio $O(nlogn)$ y el peor caso $O(n^2)$ generando que sea un algoritmo inestable, aunque dependiendo de la implementación el peor caso puede ser optimizado.

1.1.4. Heapsort

Se trata de un método de ordenamiento basado en un heap, a este algoritmo lo podemos dividir en dos partes:

- Heapify
- Heapsort

Hapify es el algoritmo que convierte cualquier arreglo de números en un heap, en este caso de máximo. Comienza desde la mitad del vector hasta el principio del mismo, chequeando que cada *raiz* sea mayor a sus dos hijos, en caso de que esto no ocurra, intercambia sus posiciones(si es un heap de máximo). Su complejidad teórica se puede calcular a través de una serie de Taylor:

$$\sum_{k=0}^{log(n)} \frac{n}{2^{k+1}} O(h) = O(n \sum_{k=0}^{log(n)} \frac{h}{2^k}) < O(n \sum_{k=0}^{\infty} \frac{h}{2^k}) \tag{6}$$

siendo h la altura del árbol con propiedad heap.

$$\sum_{k=0}^{\infty} \frac{h}{2^k} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2 \tag{7}$$

$$O(n \sum_{k=0}^{log(n)} \frac{h}{2^k}) = O(n) \tag{8}$$

Entonces la complejidad algorítmica de Heapify es $O(n)$.

Luego tenemos el otro bucle que es propiamente el ordenamiento del heap, esta parte del algoritmo se ejecuta si o si $n-1$ veces, sin importar el set de datos, pero dentro del bucle, hay una función llamada **siftdown**, que tiene como complejidad una cota de $O(20log(n))$, debido a que dentro de ella, el *while* como mucho se ejecutará $log(n)$ veces.

Para concluir, multiplicando tenemos que:

$$O(20log(n))(n-1) = O(20nlog(n)) \tag{9}$$

En el caso promedio, obviamente el bucle *while* dentro de siftdown se ejecuta menos veces que en el peor caso. Eso genera que el caso promedio este bastante por debajo de $O(20nlog(n))$ y siendo un algoritmo de ordenamiento aun más rápido que el mergesort, como veremos en la práctica.

1.1.5. Mergesort

Mergesort se trata de un algoritmo recursivo basado en la técnica de División y Conquista, estos algoritmos, como ya vimos, se caracterizan por ir dividiendo el problema en partes, resolverlo por separado para luego juntar las soluciones. Del pseudocódigo podemos observar que se tienen dos funciones:

- mergesort
- merge

En este caso, el algoritmo se llama a sí mismo 2 veces. Por lo tanto, al ser de División y Conquista,entonces para la primer llamada de la función mergesort es:

$$T_{MS}(n) = 2T_{MS}(\frac{n}{2}) + (T_m(n)) \tag{10}$$

siendo $T_m(n)$ el tiempo de ejecución de la **función merge**. Ahora analicemos este tiempo, $T_m(n)$ depende de la suma de las longitudes de los vectores que recibe, y cómo vienen ordenados cada uno. En el caso promedio el *while* se

realizará $n - i$ veces, siendo i la cantidad de elementos que le faltan a un vector para completar el vector resultante:

$$T_m(n) = O(12n - 10i) \tag{11}$$

A diferencia del peor caso que el *while* se ejecutará $n - 1$ veces, quedando solamente un elemento de algún vector para agregar al vector resultante, es decir que los 2 elementos más grandes estén en vectores separados:

$$T_m(n) = O(12(n - 1)) \tag{12}$$

A pesar de esto, las constantes que multiplican a n en ambos casos es 12, por lo tanto:

$$T_m(n) = O(12n) \tag{13}$$

Volviendo a la función **mergesort**, ésta siempre se comporta igual para cualquier tipo de sets de datos, suponiendo que $n = 2^k$ entonces:

$$T_{MS}(n) = 2^k T_{MS}(\frac{2^k}{2^k}) + kO(12n) = n + 12nlog(n) \tag{14}$$

por lo tanto finalizamos que:

$$T_{MS}(n) = O(12nlog(n)) = O(nlog(n)) \tag{15}$$

1.2. Cálculo del tiempo de ejecución

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. Este es el punto C como se hace? A que se refiere? Porque el punto siguiente D es la estimación de las corridas

Selección

La estimación se realiza a partir del resultado obtenido en la sección anterior por ejemplo para 50 números el calculo del tiempo de ejecución seria de

$$T_{50} = \frac{1}{2}(50^2 - 50) = 1225$$

Podemos indicar que estas son unidades de procesamiento un calculo más exacto nos llevaría al conteo del costo de cada operación del algoritmo. Haciendo cálculos similares nos da la tabla siguiente:

numero	tiempo = $\frac{1}{2}(n^2 - n)$
50	1225
100	4950
500	124750
1000	499500
2000	1999000
3000	4498500
4000	7998000
5000	12497500
7500	28121250
10000	49995000

Esta tabla sirve tambien para el metodo de insercion dado que da como maximo las mismas cantidad de iteraciones.

Quicksort

Podemos aproximar su tabla como $n.log(n)$

numero	tiempo = $nlog2(n)$
50	288
100	664
500	4482
1000	9965
2000	21931
3000	34652
4000	47863
5000	61438
7500	96545
10000	132877

Heapsort

El tiempo de ejecución de este algoritmo a través de los resultados obtenidos con diferentes volúmenes de datos nos da la siguiente tabla:

numero	tiempo = $20nlog_2(n)$
50	5643.8
100	13287.7
500	89657.8
1000	199315.7
2000	438631.4
3000	693044.8
4000	957262.7
5000	1228771.2
7500	1930901.2
10000	2657542.5

Mergesort

A partir de los resultados de la sección anterior, calcularemos el tiempo de ejecución mediante:

$$T_n = 12nlog(n)$$

Para los diferentes volúmenes de datos, calculando con logaritmo en base 2, obtenemos la siguiente tabla:

numero	tiempo = $12nlog(n)$
50	3386.3
100	7972.6
500	53794.7
1000	119589.4
2000	263178.8
3000	415826.9
4000	57435706
5000	737262.7
7500	1158540.7
10000	1594525.5

1.3. Estimación de los tiempos de ejecución

Selección

numero	tiempo[seg]
50	0.000203688939412
100	0.000532706578573
500	0.0109893480937
11000	0.0393036206563
2000	0.180919011434
3000	0.427456299464
4000	0.725763320923
5000	1.09040602048
7500	2.63392798106
10000	3.79089895884

Inserción

numero	tiempo[seg]
50	0.000153017044067
100	0.000789546966553
500	0.0129567146301
1000	0.0507660150528
2000	0.198131513596
3000	0.454863524437
4000	0.8265198946
5000	1.21837227345
7500	2.73131990433
10000	5.05489246845

Quicksort

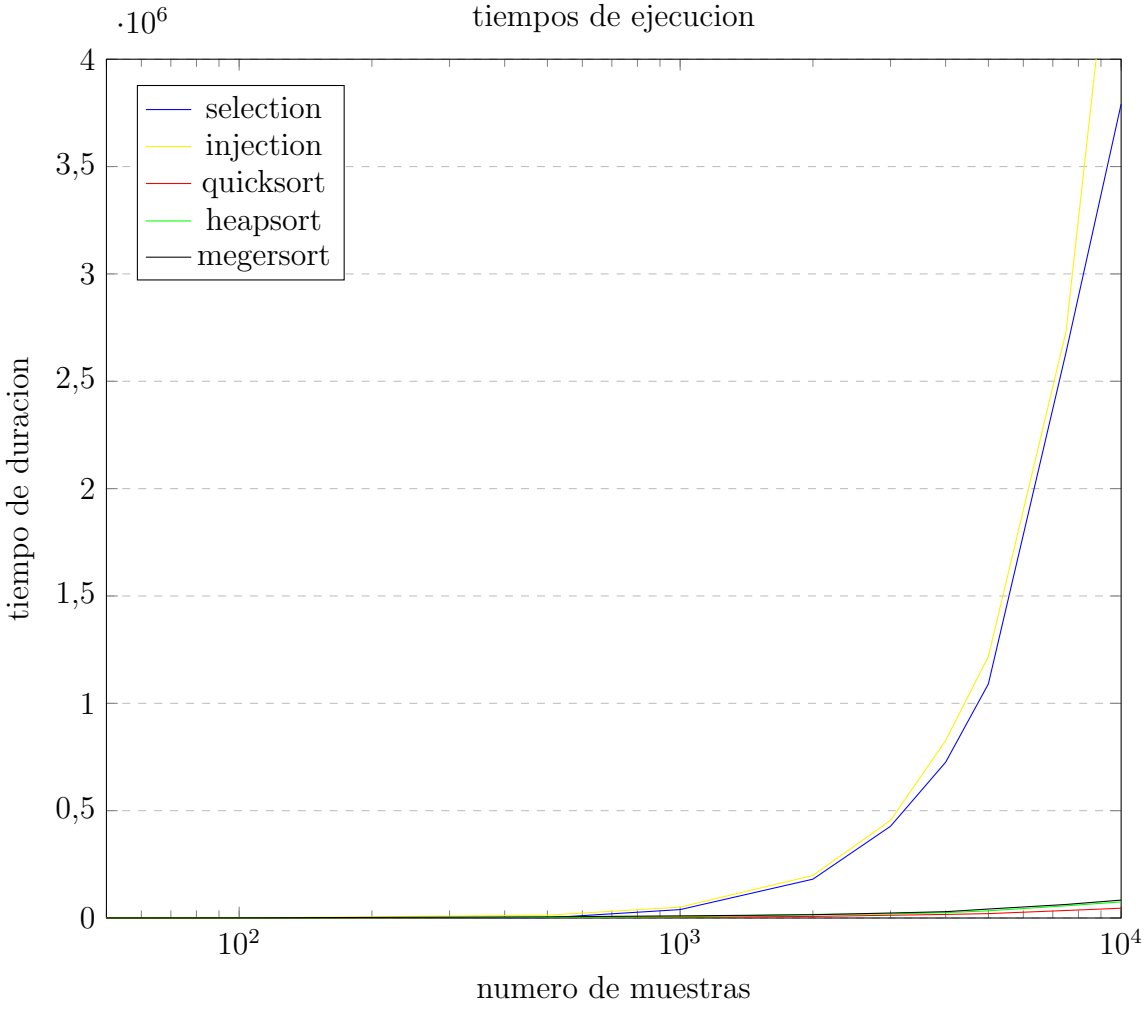
numero	tiempo[seg]
50	0.000139880180359
100	0.000377511978149
500	0.00223593711853
1000	0.00327577590942
2000	0.00749778747559
3000	0.0188936948776
4000	0.0167666196823
5000	0.0205631256104
7500	0.0351025819778
10000	0.0456968307495

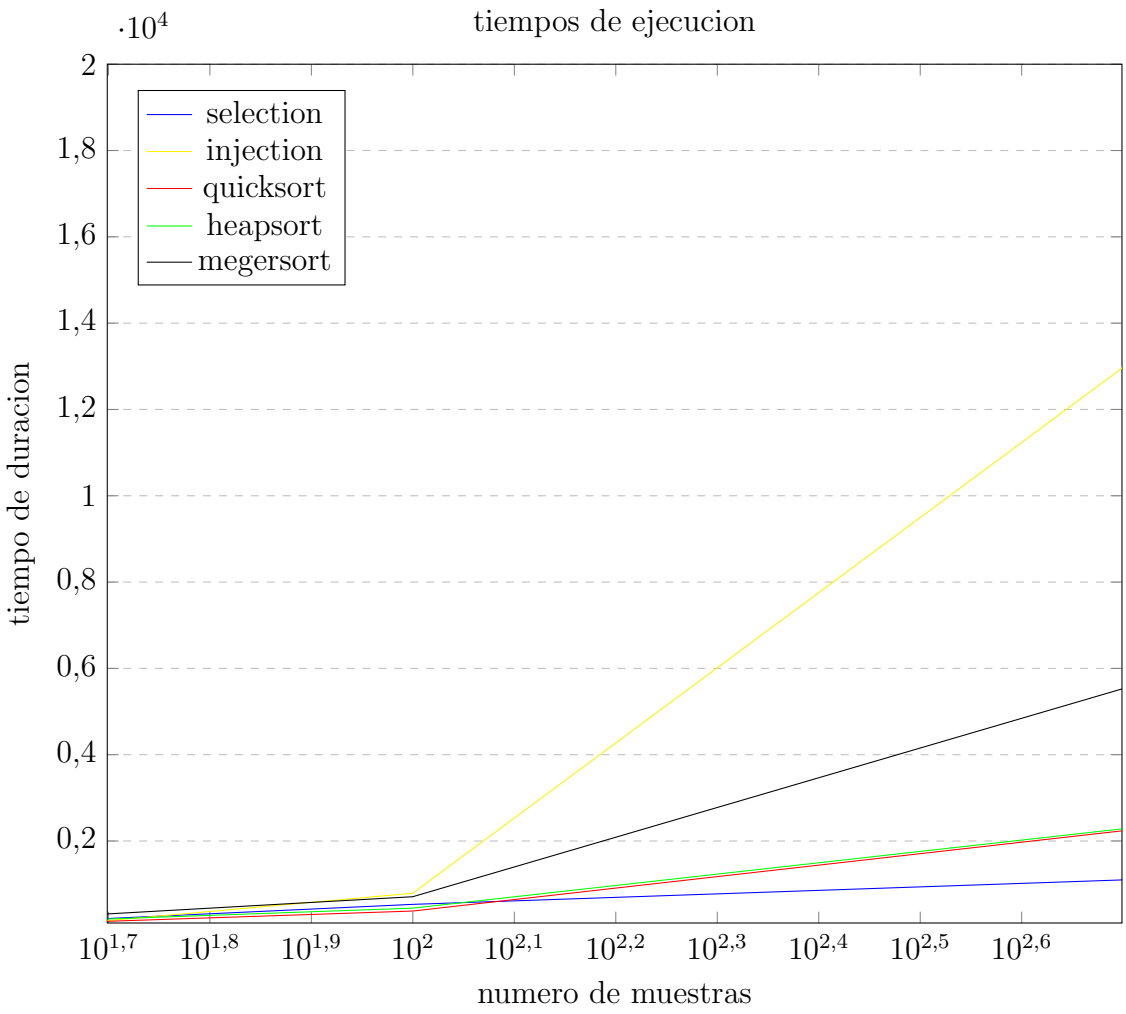
Heapsort

numero	tiempo[seg]
50	0.000192523002625
100	0.000444531440735
500	0.00228116512299
1000	0.00560901165009
2000	0.0169732093811
3000	0.019061589241
4000	0.0249751329422
5000	0.0337036848068
7500	0.0568652629852
10000	0.0751173734665

Mergesort

numero	tiempo[seg]
50	0.000310349464417
100	0.000709843635559
500	0.00552499294281
1000	0.0100009202957
2000	0.0153873205185
3000	0.0233342170715
4000	0.0295916080475
5000	0.0419291496277
7500	0.0631191015244
10000	0.0835010290146


tiempos de ejecucion
$\cdot 10^6$
4
3,5
3
2,5
2
1,5
1
0,5
0
selection
injection
quicksort
heapsort
megersort
tiempo de duracion
10^2
10^3
10^4
numero de muestras



1

1.4. Características del set para el peor caso posible

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).

Selección

Este algoritmo tiene el mismo tiempo en todos los casos con lo cual no existe un peor set. Pero un Set ordenado haría irrelevante el ordenarlo y aun así se ejecutarían todos los pasos.

Inserción

El peor caso se da cuando el set a ordenar se encuentra ordenado en forma descendente para la implementacion realizada en este trabajo. (en el caso que fuera una implementación para ordenar en forma descendente el peor caso seria entregar un set ordenado de forma ascendente). En este caso se realizarian n^2 cambios

Quicksort

Este algoritmo tiene como peor caso cuando el pivot es el menor de los numeros del conjunto o el mayor de manera tal que queden dos listas una con 0 elementos y otra con n-1. En este caso el orden es de n^2.

Heapsort

Este algoritmo tiene como peor caso cuando el vector que se ingresa ya está ordenado de la manera que se pidió. Porque el heapify siempre crea un vector inverso al orden solicitado, si piden en orden creciente, heapify crea un heap de máximo.Por lo tanto, haría la mayor cantidad de intercambios posibles.

Mergesort

Este algoritmo tiene el peor caso cuando los dos elementos más grandes del vector se encuentran en divisiones distintas, generando que el *while* se tenga que ejecutar $n - 1$ veces. Un ejemplo de esto, sería que en la primera mitad del vector estén todos los números pares y en la otra todos los números impares.

1.5. 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.

Inserción

numero	insercion	worstcase
50	0.0001530170	0.000303983688354
100	0.000789546966553	0.00102963447571
500	0.0129567146301	0.105214285851
1000	0.0507660150528	0.104745006561
2000	0.198131513596	0.605363607407
3000	0.454863524437	1.2244461298
4000	0.8265198946	1.85734820366
5000	1.21837227345	2.72190787792
7500	2.73131990433	5.73363921642
10000	5.05489246845	10.2442992687

Quicksort

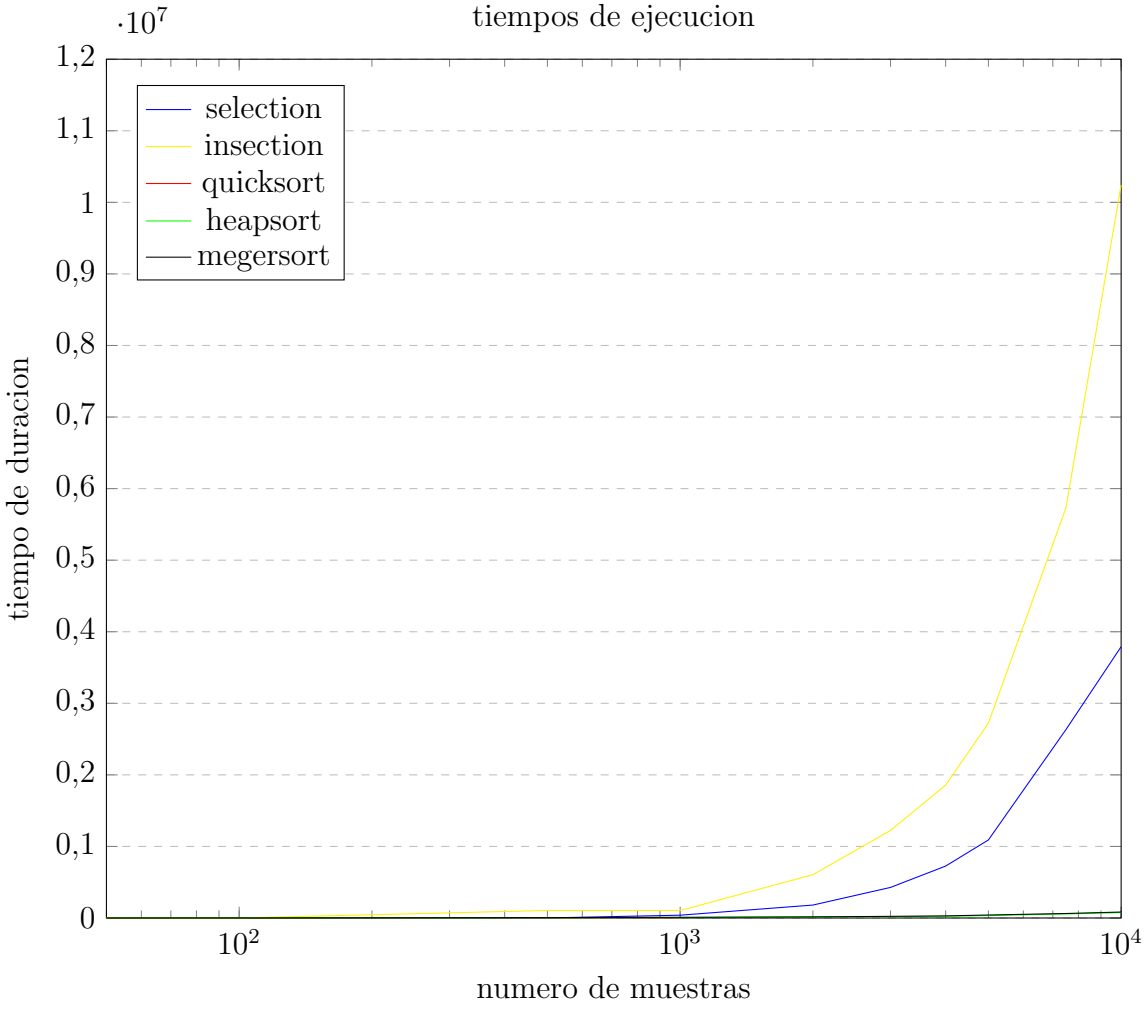
numero	quicksort[seg]	worstcase
50	0.000139880180359	0.000417041778564
100	0.000377511978149	0.00114738941193
500	0.00223593711853	0.0248492956161
1000	0.00327577590942	error maximo anidacion recursiva
2000	0.00749778747559	error maximo anidacion recursiva
3000	0.0188936948776	error maximo anidacion recursiva
4000	0.0167666196823	error maximo anidacion recursiva
5000	0.0205631256104	error maximo anidacion recursiva
7500	0.0351025819778	error maximo anidacion recursiva
10000	0.0456968307495	error maximo anidacion recursiva

Heapsort

numero	tiempo[seg]	worstcase
50	0.000192523002625	0.000309419631958
100	0.000444531440735	0.000581097602844
500	0.00228116512299	0.00287048816681
1000	0.00560901165009	0.00602440834045
2000	0.0169732093811	0.0125813961029
3000	0.019061589241	0.0207741260529
4000	0.0249751329422	0.0280975818634
5000	0.0337036848068	0.0369673728943
7500	0.0568652629852	0.058920788765
10000	0.0751173734665	0.0707016944885

Mergesort

numero	tiempo[seg]	worstcase
50	0.000310349464417	0.00034384727478
100	0.000709843635559	0.00091507434845
500	0.00552499294281	0.00383307933807
1000	0.0100009202957	0.00577623844147
2000	0.0153873205185	0.0130363941193
3000	0.0233342170715	0.0200843811035
4000	0.0295916080475	0.0264275789261
5000	0.0419291496277	0.0344556093216
7500	0.0631191015244	0.0520595788956
10000	0.0835010290146	0.0712057828903



El algoritmo de seleccion tiene como worstcase los mismo valores que en el caso random dado que la cantidad de pasos de ejecucion se mantiene constante, motivo por el cual no se realizo una corrida de este. El algoritmo de Insercion empeora sus valores, lo cual se puede observar en la tabla que presenta los dos valores el random y el worstcase. Quicksort empero sus valores hasta el caso de 1000 donde debido a restricciones de el lenguaje en que se implemento el algoritmo Python lanzo que no pudo anidar mas esto se puede salvar con un arreglo en la profundidad del stack que se debe permitir pero para los efectos del presente informe consideramos que alcanza con observar que los datos no solo empeoran sino que consumen todo el stack. Heapsort y Mergesort son los unicos casos donde no solo no empeoran sino que incluso a veces mejoran los resultados. Tanto el worstcase como el random se encuentran en el mismo orden y esto se debe a que ambos algoritmos tienen un orden $n.log(n)$ para el worstcase como para el caso random.

De estas observaciones y las recogidas en el analisis de la seccion anterior podemos comprobar que Heapsort y Mergesort tienen el comportamiento mas uniforme. Sin embargo Quicksort puede presentar tiempos de ejecucion mejores para el caso aleatorio cuando no exista orden dentro del

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

Del gráfico se puede observar que selección e inserción son algoritmos de orden cuadrático mientras que los métodos restantes son de orden logaritmicos y que para muestras mayores a 1000 sus tiempos de ejecución son mucho menores a los primeros. Heapsort, en nuestro análisis teórico, nos resultó ser más lento que el mergesort, debido a la implementación y a que esa cota se ajustaba más al peor caso que al caso promedio y una vez corridos ambos algoritmos pudimos ver que es un poco más rápido porque internamente, el bucle while, no se ejecuta tantas veces. En el caso del algoritmo de inserción a pesar que teóricamente tiene tiempos, en promedio menores que el de selección no fue el caso en nuestras pruebas sino todo lo contrario. Esto se debe a que los swaps realizados por el inserción pesaron más que la menor cantidad de comparaciones frente al de selección.
Comparando respecto al caso teórico los resultados son coincidentes, para valores grandes los algoritmos se vuelven asintóticos y pondera más su orden que las constantes que los acompañan.

1.7. Instrucciones para la ejecucion de los algoritmos de ordenamiento

Ingresar el comando siguiente:

```
1 python sorts.py -i algoritmo -n cantidad
```

donde algoritmo puede ser alguno de los siguientes = selection, insertion, quicksort, heapsort, mergesort
y cantidad se corresponde con las cantidades solicitadas para este trabajo es decir 50,100,500,1000,2000,3000,4000,5000,7500,10000
De esta manera si ejecutara el caso con datos random.

Para el worstcase agregarle la opcion -w por ejemplo para 50 muestras y el algoritmo mergesort en el worstcase.

```
1 python sorts.py -i algoritmo -n cantidad -w
```

La ejecucion de los comando lanzara 10 valores de tiempo de ejecucion expresados en segundos seguidos por un ultimo tiempo indicando el tiempo medio.

1.8. Codigo Algoritmos de ordenamiento

```
import sys

def selectionsort(a):
  for j in xrange(len(a)):
    iMin=j
    for i in xrange(j+1,len(a)):
      if a[i] < a[iMin]:
        iMin=i

    if iMin!=j:
      aux = a[j]
      a[j]=a[iMin]
      a[iMin]=aux
    # print(a[:j])
def insertionsort(a):
    for i in xrange(1, len(a)):
        j = i-1
        key = a[i]
        while (a[j] > key) and (j >= 0):
           a[j+1] = a[j]
           j -= 1
        a[j+1] = key

def quicksort(arr):
    less = []
    pivotList = []
    more = []
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        for i in arr:
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
```

```
            else:
                pivotList.append(i)
        less = quicksort(less)
        more = quicksort(more)
        return less + pivotList + more

def heapsort(lst):
  ''' Heapsort. Note: this function sorts in-place (it mutates the
      list). '''

  # in pseudo-code, heapify only called once, so inline it here
  for start in range((len(lst)-2)/2, -1, -1):
    siftdown(lst, start, len(lst)-1)

  for end in range(len(lst)-1, 0, -1):
    lst[end], lst[0] = lst[0], lst[end]
    siftdown(lst, 0, end - 1)
  return lst

def siftdown(lst, start, end):
  root = start
  while True:
    child = root * 2 + 1
    if child > end: break
    if child + 1 <= end and lst[child] < lst[child + 1]:
      child += 1
    if lst[root] < lst[child]:
      lst[root], lst[child] = lst[child], lst[root]
      root = child
    else:
      break


def mergesort(array):
  if len(array) < 2:
    return array

  middle = int(len(array)/2)
  left = mergesort(array[:middle])
  right = mergesort(array[middle:])

  return merge(left, right)

```

```
79  def merge(left, right):
80      result = []
81      left_idx, right_idx = 0, 0
82      while left_idx < len(left) and right_idx < len(right):
83          # change the direction of this comparison to change the
                direction of the sort
84          if left[left_idx] <= right[right_idx]:
85              result.append(left[left_idx])
86              left_idx += 1
87          else:
88              result.append(right[right_idx])
89              right_idx += 1
90
91      if left_idx < len(left):
92          result.extend(left[left_idx:])
93      if right_idx < len(right):
94          result.extend(right[right_idx:])
95      return result
96
97
98  def sort(type,number,worstcase):
99
100   sorts = {'selection': selectionsort,
101   'insertion': insertionsort,
102   'quicksort': quicksort,
103   'mergesort': mergesort,
104   'heapsort': heapsort}
105
106   method_name = myargs['-i']
107      # set by the command line options
108     #if myargs[]
109   if type not in sorts:
110     raise Exception("Sort Algorithm %s not implemented" % type)
111     exit()
112
113   if worstcase:
114     file=number+'.'+type+'.worstcase'
115   else:
116     file=number+'.random'
117   b=[]
118   with open(file, 'r') as f:
119     for line in f:
120       b.append(int(line))
```

```
121   #print(b)
122   from timeit import default_timer as timer
123   #slice create a copy of the array
124   results={}
125   mean=0
126   for x in xrange(0,10):
127     print(x)
128     a=b[:]
129     start = timer()
130     sorts[type](a)
131     end = timer()
132     results[x]=end-start
133     print (results[x])
134     mean+=end-start
135   print(x)
136   mean=mean/(x+1)
137   print (mean)
138   return
139
140 def getopts(argv):
141     opts = {} # Empty dictionary to store key-value pairs.
142     while argv: # While there are arguments left to parse...
143         if argv[0][0] == '-': # Found a "-name value" pair.
144           if argv[0][1] != 'w':
145             opts[argv[0]] = argv[1] # Add key and value to the
                  dictionary.
146           else:
147             opts['-w']='y'
148         argv = argv[1:] # Reduce the argument list by copying it
              starting from index 1.
149     return opts
150
151 if __name__ == '__main__':
152     from sys import argv
153     myargs = getopts(argv)
154     if myargs.has_key('-i'):
155       sort(myargs['-i'],myargs['-n'],myargs.has_key('-w'))
156     exit()
```

2. Parte 2: 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.

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.

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

```
1  #!/usr/bin/python
2  import time
3  import random
4  import sys
5
6  FREE_POSITION = -1 #representa una posicion libre en un equipo
7
8  #Dimensiones del problema:
9  teams_amount, players_by_team, players_amount = 20, 10, 200
10 #teams_amount, players_by_team, players_amount = 3, 2, 6
11
12 def load_players_peferences():
13    players_preferences = [range(teams_amount) for y in
          range(players_amount)]
14    for i in range(players_amount):
15       file_name =
             './setDePruebasParte2/'+'jugador_['+str(i+1)+'].prf'
16       with open(file_name, 'r') as preference_list:
17          pl = [int(j) for j in list(preference_list)]
```

```
18          players_preferences[i] = pl
19      return players_preferences
20
21  def load_teams_preferences():
22      teams_preferences = [range(players_amount) for y in
            range(teams_amount)]
23      for i in range(teams_amount):
24          file_name =
                './setDePruebasParte2/'+'equipo_['+str(i+1)+'].prf'
25          with open(file_name, 'r') as preference_list:
26              pl = [int(j) for j in list(preference_list)]
27              teams_preferences[i] = pl
28      return teams_preferences
29
30  #Genera un set de archivos de prueba en el directorio
        setDePruebasParte2
31  #Para equipos: equipo_[nro].prf => generate_test_set(teams_amount,
        players_amount, 'equipo')
32  #Para jugadores: jugador_[nro].prf =>
        generate_test_set(players_amount, teams_amount, 'jugador')
33  def generate_test_set(files_amount, lines_amount,
        file_name_prefix):
34      for i in range(files_amount):
35          file_name =
                './setDePruebasParte2/'+file_name_prefix+'_['+str(i+1)+'].prf'
36          fh = open(file_name,'w')
37          members_ranking = range(lines_amount)
38          random.shuffle(members_ranking)
39          for j in members_ranking:
40            fh.write("%s\n" % (j+1))
41          fh.close()
42
43  def init_teams():
44      teams = [[FREE_POSITION for x in range(players_by_team)] for y
            in range(teams_amount)]
45      return teams
46
47  def team_vacancies_available(team):
48      return FREE_POSITION in team
49
50  def add_player_to_team(team, player_number):
51      team[team.index(FREE_POSITION)] = player_number
52
```

```
53 def remove_player_from_team(team, player_number):
54    team[team.index(player_number)] = FREE_POSITION
55
56 def move_player(from_team, to_team, player_number):
57    remove_player_from_team(from_team, player_number)
58    add_player_to_team(to_team, player_number)
59
60 def compare_preferences(preferences, a_number, b_number):
61    a_rank = preferences.index(a_number)
62    b_rank = preferences.index(b_number)
63    if(a_rank < b_rank):
64       return 1
65    elif (a_rank > b_rank):
66       return -1
67    else:
68       return 0
69
70 def find_player(teams, player_number):
71    for team_number in range(teams_amount):
72       if player_number in teams[team_number]:
73          return team_number
74    return FREE_POSITION
75
76 def run_gale_shapley(teams, teams_preferences,
     players_preferences):
77    vacancies_available = teams_amount * players_by_team
78    team_vacancies_available = [players_by_team for x in
        range(teams_amount)]
79    while vacancies_available > 0:
80       for team_number in range(teams_amount):
81          while team_vacancies_available[team_number]:
82             team_preference = teams_preferences[team_number].pop(0)
83             other_team_number = find_player(teams, team_preference)
84             if(other_team_number == FREE_POSITION):
85                add_player_to_team(teams[team_number],
                      team_preference)
86                vacancies_available -= 1
87                team_vacancies_available[team_number] -= 1
88             else:
89                if(compare_preferences(players_preferences[team_preference-1],
                      team_number+1, other_team_number+1) > 0):
90                   move_player(teams[other_team_number],
                         teams[team_number], team_preference)
```

```
91                  team_vacancies_available[other_team_number] += 1
92                  team_vacancies_available[team_number] -= 1
93
94  def print_preferences(preferences):
95     preferences_str = [str(x) for x in preferences]
96     for i in range(len(preferences_str)):
97        sys.stdout.write(str(i) + ":")
98        print preferences[i]
99        sys.stdout.flush()
100
101 def print_everything(teams_preferences, players_preferences,
        teams) :
102    print "Preferencias de los equipos:"
103    print_preferences(teams_preferences)
104
105    print "Preferencias de los jugadores:"
106    print_preferences(players_preferences)
107
108    print "Equipos formados:"
109    print_preferences(teams)
110
111    sys.stdout.flush()
112    time.sleep(0.1)
113
114 def main():
115    print "TP1 - Parte 2: Algoritmo de Gale Shapely"
116
117 #  generate_test_set(teams_amount, players_amount, 'equipo')
118 #  generate_test_set(players_amount, teams_amount, 'jugador')
119
120    teams_preferences = load_teams_preferences()
121    players_preferences = load_players_peferences()
122    teams = init_teams()
123
124    run_gale_shapley(teams, teams_preferences, players_preferences)
125
126    print_everything(teams_preferences, players_preferences, teams)
127
128    return teams
129
130
131 #Genera la tabla de jugadores y equipos en LaTeX para el informe
132 def to_latex(matrix):
```

```
133     column_definition = "|".join(["c" for x in
            range(len(matrix[0]))])
134     print("\\begin{center}")
135     print("\\begin{longtable}{ |" + column_definition + "| } ")
136     print(" \\hline")
137 
138     for row in range(len(matrix)):
139        sys.stdout.write(" & ".join(map(str, matrix[row])) + "
               \\\\\n")
140 
141     print(" \\hline")
142     print("\\end{longtable}")
143     print("\\end{center}")
144 
145 def players_preferences_to_latex():
146     preferences = load_players_peferences()
147     matrix = [range(len(preferences))] + zip(*preferences)
148     matrix = zip(*matrix)
149     to_latex(matrix)
150 
151 def teams_preferences_to_latex():
152     preferences = load_teams_preferences()
153     matrix = [range(len(preferences))] + zip(*preferences)
154     to_latex(matrix)
155 
156 def results_to_latex():
157     results = main()
158     matrix = [range(len(results))] + zip(*results)
159     to_latex(matrix)
160 
161 if __name__ == "__main__":
162     main()
```

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

2.2.1. Órden del algoritmo

El algoritmo es de orden O($3E^2J + EJ^2 + 12EJ$).

Comencemos nuestro análisis viendo cuantas veces se repite el bloque interno en el peor caso. Llamaremos "bloque interno.ª las líneas entre la 82

y la 92:

```
82              team_preference = teams_preferences[team_number].pop(0)
83              other_team_number = find_player(teams, team_preference)
84              if(other_team_number == FREE_POSITION):
85                 add_player_to_team(teams[team_number],
                       team_preference)
86                 vacancies_available -= 1
87                 team_vacancies_available[team_number] -= 1
88              else:
89                 if(compare_preferences(players_preferences[team_preference-1],
                       team_number+1, other_team_number+1) > 0):
90                    move_player(teams[other_team_number],
                          teams[team_number], team_preference)
91                    team_vacancies_available[other_team_number] += 1
92                    team_vacancies_available[team_number] -= 1
```

Sea J la cantidad de jugadores y E la cantidad de equipos. Sea una función P(t) que represente la cantidad de pares (x,y) tales que el equipo x le haya ofrecido una posición al jugador y (independientemente de la posición que ocuparía y en el equipo) al término de la iteración t del bloque interno. Entonces $P(t+1) > P(t)$ para todo t ya que un equipo no ofrece 2 veces al mismo jugador. Pero P() al final de todas las iteraciones puede ser a lo sumo igual a todas las combinaciones posibles de equipo-jugador, por lo tanto habrán a lo sumo E*J iteraciones del bloque interno.

Ahora analicemos el bloque interno en si. En el peor caso, la cantidad máxima posible de operaciones ejecutadas en este bloque ocurre cuando el primer if es evaluado en falso y el segundo es evaluado en verdadero. En ese caso el total de operaciones es de 3E+J+9 (suma de las operaciones indicadas a continuación en negrita). Pero como por lo visto anteriormente este bloque se repite a lo sumo EJ veces, entonces el total de operaciones es de $3E^2J + EJ^2 + 9EJ$.

- Línea 81. La evaluación de la condición del while. **1 op.**
- Línea 82. La remoción del elemento. **1 op.**
- Línea 83. La búsqueda de un jugador en los equipos. En la función find_player, la condición del for se evalúa como máximo E veces, lo cual implica E operaciones. El in del if implica como máximo J/E comparaciones y como por el for que lo envuelve, el if se repite como máximo E veces este if implica J operaciones. **E+J ops.**
- línea 84. La evaluación de la condición del if. **1 op.**

- Si la condición es verdadera
  - Línea 85. Agregar un elemento a una lista. 1 op.
  - Línea 86. Decrementar una variable. 1 op.
  - Línea 87. Decrementar una variable. 1 op.
- Si la condición es falsa
  - Línea 89. Comparar las preferencias del jugador. En las primeras dos líneas de la función compare_preferences llamados a la función index que implican como máximo E comparaciones cada una, y los if que siguen como máximo pueden hacer 2 comparaciones. **2E+2 ops.**
  - Si la condición es verdadera:
    - Línea 90. Mover un elemento de una lista a otra (quitar + agregar). **2 ops.**
    - Línea 91. Incrementar una variable. **1 op.**
    - Línea 92. Decrementar una variable. **1 op.**

Finalmente vamos a analizar las estructuras de control que rodean al bloque interno:

```
79  while vacancies_available > 0:
80      for team_number in range(teams_amount):
81          while team_vacancies_available[team_number]:
82          [bloque interno]
```

Como hemos visto, el bloque interno se puede repetir como máximo EJ veces. Por lo tanto, la evaluación de las condiciones de las estructuras de control que lo rodean también pueden repetirse como máximo EJ veces. Como son 3 evaluaciones de condiciones, es un total de $3EJ$ operaciones.

- Línea 79. La evaluación de la condición del while. **EJ ops.**
- Línea 80. Extracción de un equipo de la lista de equipos. **EJ ops.**
- Línea 81. La evaluación de la condición del while. **EJ ops.**

Por lo tanto, el total de operaciones del algoritmo de Gale-Sapley implementado es de $3E^2J + EJ^2 + 12EJ$.

2.2.2. El algoritmo siempre devuelve un matching estable

Lo probamos por reducción al absurdo. Supongamos que el algoritmo devuelve 2 tuplas $(e, j, j_2, j_3, j_4...)$ y $(e', j', j_a, j_b, j_c, ...)$ tales que: el equipo e prefiere al jugador j' antes que a j y el jugador j' prefiere al equipo e antes que a e'

¿Es posible que e le haya ofrecido a j' antes que a j?

Si no fue así, entonces j debió estar antes que j' en la lista de preferencia de e, lo cual contradice la suposición inicial de que e prefiere a j'.

Si fue así, entonces j' rechazó a e por algún mejor equipo e". Entonces o bien $e'' = e'$, lo cual quiere decir que j' prefirió al equipo e' por sobre e. O bien $e'' \neq e'$ y entonces por transitividad e' fue preferido por j' por sobre e. Ambas posibilidades contradicen la suposición inicial de que el jugador j' prefiere al equipo e antes que a e'.

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

Para poder probar el algoritmo hemos construido un set de pruebas consistente en archivos que contienen las preferencias de los jugadores y de los equipos. Las tablas que se proveen a continuación representan el contenido de estos archivos, y la última tabla es el resultado de la ejecución del algoritmo.

2.3.1. Preferencias de los equipos

En la siguiente tabla, las columnas representan los equipos numerados del 0 al 19. Cada uno tiene el listado de los 200 jugadores ordenados de arriba a abajo por preferencia desde el más preferido al menos preferido.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	147	173	136	189	102	186	78	152	146	108	145	79	149	159	138	59	81	43	102
158	37	151	11	4	191	72	92	196	185	59	35	5	83	165	171	134	129	4	80
146	105	136	179	78	184	110	39	53	87	100	34	76	100	119	77	99	185	156	24
65	179	53	128	176	158	123	178	24	141	162	23	191	162	105	184	110	84	125	64
131	84	66	185	75	183	3	16	28	34	74	57	96	146	115	85	96	65	98	58
171	49	68	15	147	39	131	83	55	151	113	109	120	19	76	66	198	100	154	154
44	176	149	42	174	83	36	173	34	107	129	119	99	98	64	157	72	25	21	16
54	61	101	192	27	96	56	3	189	23	116	16	155	67	95	48	177	166	20	36
66	119	165	199	81	20	31	52	9	153	151	196	186	1	187	56	51	66	122	155
190	114	117	156	61	175	139	45	23	83	35	28	181	163	15	13	40	176	183	158
101	149	86	40	151	16	40	70	31	28	67	112	187	155	46	45	136	49	84	130
132	180	147	59	184	3	100	199	71	169	156	188	57	32	150	3	138	138	96	10
109	124	99	200	40	33	76	116	118	55	49	1	134	148	124	49	114	96	75	110
117	98	119	34	47	32	17	32	44	133	37	15	7	90	32	185	104	95	53	169
72	104	49	41	180	126	12	37	121	178	159	95	160	71	170	136	132	91	178	39
175	96	18	163	43	12	133	101	16	71	63	48	11	68	117	72	11	134	81	183
80	186	43	110	120	149	49	150	93	65	125	96	112	158	55	113	50	55	11	23
199	162	59	16	32	140	67	185	70	48	112	91	42	124	45	129	165	175	52	38
187	177	185	50	46	14	175	134	108	148	29	134	48	160	24	115	166	174	167	198
36	195	128	100	50	90	74	33	144	198	90	136	108	12	36	186	161	106	97	166
94	1	58	113	74	129	38	153	132	100	91	184	171	115	147	71	73	193	121	88
184	103	32	66	155	107	2	17	92	155	6	6	74	11	175	188	123	115	29	25
103	160	190	186	117	42	69	159	123	93	165	155	77	23	199	150	122	40	137	48
81	68	192	159	26	63	116	104	164	131	86	116	41	61	158	68	1	23	143	63
165	38	166	114	42	57	173	34	4	195	105	198	73	37	26	91	42	128	115	114

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

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

147	65	79	70	134	73	6	193	111	165	195	59	40	89	138	112	152	75	5	37
159	31	113	31	19	119	55	135	140	111	75	163	34	190	75	130	193	88	2	31
197	10	39	14	163	200	41	141	68	64	71	193	75	151	43	175	190	117	80	40
176	23	156	160	170	181	150	73	58	73	57	115	195	17	18	78	167	68	89	75
127	2	159	142	139	99	122	131	161	86	148	79	8	36	161	53	8	195	169	179
188	118	191	120	24	26	141	192	160	170	101	39	72	96	21	74	62	47	57	76
118	87	64	92	13	185	164	115	42	77	103	199	26	62	54	21	94	90	141	108
130	183	109	124	141	163	178	181	188	114	121	168	104	87	172	58	164	54	126	45
82	144	172	75	18	190	89	191	90	158	33	158	169	197	128	122	184	73	128	2

2.3.2. Preferencias de los jugadores

En la siguiente tabla, las filas representan los jugadores numerados del 0 al 199. Cada uno tiene el listado de los 20 equipos ordenados de izquierda a derecha por preferencia desde el más preferido al menos preferido.

0	2	17	16	14	18	3	19	6	10	20	13	15	8	7	11	1	5	12	9	4
1	18	12	15	9	19	7	13	1	4	3	20	2	5	11	14	6	16	10	8	17
2	17	9	10	11	18	7	19	16	1	12	3	5	2	6	13	20	4	14	8	15
3	14	4	9	12	18	3	20	11	8	15	17	2	19	16	1	5	6	13	10	7
4	10	19	18	3	14	16	20	2	8	15	13	11	17	4	7	12	6	9	5	1
5	12	11	18	1	20	15	10	4	19	8	9	13	5	17	16	14	7	6	3	2
6	20	3	10	8	9	7	14	17	4	15	11	16	13	12	18	2	6	19	1	5
7	1	19	8	12	7	6	13	14	16	3	9	4	2	11	5	15	10	20	18	17
8	12	8	19	14	2	4	11	18	6	20	7	1	3	10	15	16	13	17	5	9
9	16	13	7	6	11	19	8	5	1	18	9	4	15	20	10	12	14	2	17	3
10	8	19	12	7	5	17	15	3	6	18	9	2	14	13	10	1	11	20	4	16
11	2	13	9	14	1	17	11	5	18	20	16	15	6	8	4	3	19	10	12	7
12	6	12	5	13	8	16	3	2	14	20	10	15	9	18	19	1	7	11	4	17
13	9	17	6	3	14	7	15	19	11	16	2	4	13	12	10	1	20	18	5	8
14	1	12	3	18	20	8	19	11	13	10	14	17	4	9	7	5	6	2	15	16
15	1	11	12	14	17	2	3	10	16	4	19	5	9	13	18	20	6	7	8	15
16	17	19	14	13	2	20	12	4	8	15	11	18	10	1	9	6	3	5	7	16
17	3	6	13	10	17	18	9	16	4	11	14	7	1	8	20	19	2	15	5	12
18	8	13	3	7	17	2	18	5	14	19	1	20	9	10	12	11	16	4	15	6
19	20	9	17	10	2	19	1	14	8	4	11	18	5	13	16	3	7	6	12	15
20	14	3	5	19	10	12	8	1	9	7	18	15	20	16	17	6	2	13	11	4
21	11	12	14	13	17	10	7	9	19	6	20	2	4	15	5	8	1	3	18	16
22	11	6	20	14	15	10	9	2	13	16	12	1	7	19	4	17	5	18	3	8
23	11	10	6	14	17	15	19	5	4	13	12	18	3	1	7	8	20	9	2	16
24	20	19	3	6	9	8	4	15	14	7	12	11	16	2	5	13	10	18	17	1
25	15	9	1	8	18	19	7	2	10	11	14	6	16	17	12	3	5	4	20	13
26	13	12	7	16	11	1	9	14	20	5	19	3	6	18	2	15	10	8	4	17
27	17	5	4	6	11	10	7	13	15	9	19	3	8	2	20	14	18	1	16	12
28	8	18	7	12	13	11	5	16	9	1	14	10	17	2	19	6	15	20	4	3
29	1	4	10	9	20	11	16	8	3	17	2	12	5	19	15	6	13	7	14	18
30	9	1	8	6	13	2	10	5	20	16	14	7	18	19	4	11	3	17	12	15
31	4	15	10	8	18	13	1	14	7	19	16	11	2	20	3	6	17	9	5	12
32	18	13	1	6	8	5	7	14	10	19	16	12	15	17	4	9	11	20	2	3
33	15	9	1	12	3	5	16	19	17	10	14	11	4	8	6	20	13	18	7	2
34	14	8	16	5	11	19	10	13	1	12	18	15	4	7	20	3	6	17	2	9
35	12	11	2	6	14	16	19	13	3	1	9	10	15	5	7	20	18	4	17	8
36	12	7	13	15	4	11	14	18	6	5	9	19	10	1	8	3	20	17	2	16
37	7	6	4	19	8	13	17	18	16	2	9	5	11	20	1	10	14	12	15	3
38	14	19	11	5	6	17	8	18	13	10	4	9	2	16	15	7	12	20	1	3
39	2	11	18	19	14	1	5	13	4	17	10	9	16	3	7	20	15	12	8	6
40	20	17	9	12	11	19	13	16	10	4	15	18	3	2	7	14	1	5	6	8
41	14	1	11	8	4	9	16	12	5	10	19	13	15	3	2	18	20	17	6	7
42	4	14	2	19	12	13	8	6	17	11	5	18	10	16	7	15	3	9	20	1
43	16	15	14	1	5	20	17	3	6	7	9	19	8	12	13	11	18	4	10	2
44	12	9	2	7	5	16	18	20	17	10	6	1	13	11	8	4	3	19	15	14
45	17	16	1	18	7	9	14	2	19	3	4	15	20	6	13	5	12	8	10	11
46	6	14	5	1	7	19	9	16	3	12	18	15	2	20	4	17	11	13	8	10
47	10	3	19	9	2	8	18	13	17	6	15	12	20	5	7	1	11	14	4	16
48	13	16	12	4	17	11	6	9	8	1	14	3	19	7	20	5	2	18	10	15
49	17	4	6	13	18	3	11	19	9	10	7	2	1	12	15	14	8	20	5	16
50	19	9	8	12	5	18	6	2	11	14	13	3	10	4	15	17	20	1	7	16
51	11	10	17	4	8	2	20	14	15	9	6	5	18	1	16	3	19	12	13	7
52	15	4	2	18	8	10	13	16	6	19	11	17	7	14	9	1	20	12	3	5
53	5	14	16	19	11	8	18	3	17	10	1	7	6	12	2	20	13	15	4	9
54	3	1	4	17	11	20	13	2	12	8	16	6	18	14	15	7	10	5	19	9

55	2	6	9	7	8	4	5	19	15	20	11	14	3	16	13	1	18	10	17	12
56	19	17	8	4	16	9	5	14	13	10	11	7	20	15	6	2	18	3	12	1
57	20	10	17	1	3	8	16	2	13	19	14	4	5	7	9	11	15	6	12	18
58	1	12	18	2	9	4	6	15	7	5	17	10	16	13	14	3	11	19	8	20
59	15	18	1	12	11	3	13	17	2	14	9	4	6	16	10	5	8	19	20	7
60	7	5	15	4	3	8	19	13	17	14	18	12	16	1	10	2	11	9	20	6
61	5	18	16	3	19	15	13	20	17	4	11	9	8	7	10	1	2	12	6	14
62	13	18	8	12	14	5	20	9	19	11	2	4	1	10	16	15	7	17	3	6
63	1	15	20	17	3	18	9	4	7	5	2	16	12	19	6	13	8	10	14	11
64	6	15	14	19	5	10	12	18	13	11	17	7	4	9	20	8	1	2	16	3
65	7	14	16	6	19	18	2	3	4	8	13	10	20	9	11	5	1	17	12	15
66	10	7	17	2	3	4	8	19	14	18	13	1	12	9	11	16	15	20	6	5
67	7	9	15	2	13	4	11	12	1	14	20	5	17	16	18	19	6	3	10	8
68	8	14	3	17	10	19	7	20	12	6	1	15	2	5	11	18	4	13	9	16
69	8	1	3	6	14	19	4	15	9	2	10	5	13	11	20	16	17	18	7	12
70	1	4	14	11	13	7	3	10	8	19	20	2	16	6	5	12	15	9	17	18
71	18	9	8	13	20	11	17	3	6	14	4	15	16	12	19	10	1	7	5	2
72	13	1	10	4	12	5	2	11	20	14	17	16	7	18	19	15	8	9	3	6
73	17	14	10	3	1	11	6	8	19	18	5	2	4	13	9	7	15	20	12	16
74	20	10	3	7	5	8	13	2	19	9	18	17	12	1	15	14	4	6	11	16
75	9	2	8	1	15	19	13	14	3	18	6	4	17	5	10	11	7	16	20	12
76	13	9	2	3	18	6	14	11	1	16	15	17	20	12	7	5	10	4	8	19
77	13	8	7	17	4	2	14	15	3	18	12	11	9	10	19	16	5	6	1	20
78	12	15	4	10	8	6	3	17	14	9	5	11	18	2	20	1	7	19	13	16
79	10	6	8	3	5	4	19	9	16	20	14	1	2	17	12	13	11	18	7	15
80	11	19	1	18	12	13	10	14	4	8	2	17	9	16	20	7	6	3	15	5
81	19	2	20	1	18	5	15	8	16	7	6	3	10	9	14	13	4	12	11	17
82	5	6	9	17	15	14	2	10	3	12	8	16	19	18	4	20	11	1	7	13
83	18	11	12	7	16	8	20	17	19	13	1	6	15	2	3	10	14	9	5	4
84	3	4	9	14	2	10	20	17	18	12	15	7	13	1	19	8	16	6	11	5
85	10	18	13	1	8	9	7	12	3	6	4	2	11	16	19	20	17	15	14	5
86	11	10	19	18	15	6	8	9	17	7	4	12	20	13	16	14	1	2	3	5
87	9	20	13	18	6	7	11	3	2	8	4	16	15	19	10	5	17	1	12	14
88	18	16	10	14	20	2	13	9	19	5	4	7	12	11	15	8	17	1	6	3
89	5	15	10	13	9	11	18	19	6	20	1	7	17	8	12	16	14	2	4	3
90	17	9	16	8	20	18	19	2	7	6	13	10	12	11	3	4	15	1	14	5
91	17	9	5	6	11	18	19	13	20	1	8	4	7	10	16	15	12	2	3	14
92	4	10	8	11	16	5	18	7	2	14	13	3	19	6	15	1	12	9	20	17
93	5	11	14	3	16	19	2	7	1	9	10	12	8	20	18	4	17	15	6	13
94	7	17	15	2	8	1	10	9	4	11	18	3	13	12	16	6	14	5	19	20
95	9	3	16	1	12	4	20	8	14	7	11	10	6	5	17	2	13	15	19	18
96	15	11	8	9	6	3	17	4	20	2	5	14	1	13	7	12	19	16	10	18
97	20	17	13	5	8	19	16	9	4	14	7	3	10	18	6	2	12	15	1	11
98	1	3	2	17	14	9	15	6	5	8	20	7	12	13	18	4	19	11	16	10
99	16	19	1	10	5	7	2	13	9	17	18	20	14	12	15	11	4	8	3	6
100	20	17	11	19	3	1	2	5	8	9	4	14	13	7	18	16	15	10	6	12
101	17	9	18	2	12	5	11	14	13	7	19	20	16	8	6	15	3	4	10	1
102	11	13	2	14	1	12	4	10	3	5	20	18	6	8	15	7	9	17	16	19
103	6	15	19	11	12	18	10	13	8	9	5	2	3	17	7	4	1	16	20	14
104	4	7	16	6	19	10	20	12	15	5	1	17	13	14	11	18	8	9	2	3
105	10	3	18	9	4	5	20	14	6	1	11	12	19	15	7	2	8	17	16	13
106	1	18	4	13	5	11	3	16	12	8	6	7	2	10	15	14	17	19	9	20
107	9	7	19	6	13	16	2	17	1	11	10	5	12	15	18	4	3	20	8	14
108	1	12	17	5	9	15	3	18	11	8	14	7	16	2	19	10	4	6	13	20
109	12	8	17	3	16	11	13	9	6	18	19	4	1	10	2	7	15	20	14	5
110	15	4	20	9	12	11	16	19	6	18	3	7	2	8	13	14	10	1	5	17
111	10	19	7	5	11	2	8	1	3	16	4	6	12	9	17	20	14	15	18	13
112	17	13	14	5	15	11	20	4	7	1	12	9	8	16	10	6	18	2	3	19
113	19	10	17	4	15	2	11	14	5	20	9	16	1	6	8	12	18	13	7	3
114	19	7	14	8	20	4	3	17	16	5	6	9	12	13	11	1	10	15	18	2
115	9	19	4	3	1	16	13	11	14	12	5	7	6	15	17	8	18	2	20	10
116	7	6	17	18	13	8	2	5	9	3	20	16	14	4	1	10	15	19	12	11
117	17	15	1	4	7	18	11	16	9	14	19	12	3	13	2	6	5	20	8	10
118	5	19	3	17	8	18	9	15	11	1	14	7	16	4	20	10	13	2	6	12
119	5	18	6	20	3	15	4	14	10	8	11	17	1	7	2	19	16	9	12	13
120	9	3	1	15	12	16	11	6	10	20	14	18	5	2	19	13	4	8	17	7
121	6	1	4	9	15	18	14	19	8	2	7	3	10	17	11	12	20	13	5	16
122	10	3	6	16	11	5	18	1	17	9	4	12	7	19	2	15	20	14	13	8
123	5	17	12	3	16	9	6	4	2	14	20	1	7	13	15	8	19	11	18	10
124	10	12	6	8	7	14	3	19	18	15	4	17	20	11	13	9	1	5	2	16
125	13	20	10	16	19	18	11	1	7	9	15	12	2	6	14	8	5	17	4	3
126	19	5	4	7	20	13	17	8	16	9	1	14	6	12	3	10	2	11	15	18
127	15	4	3	11	10	19	14	9	5	16	18	20	8	1	13	7	12	2	17	6
128	16	7	9	20	6	19	11	17	14	1	12	5	3	15	10	13	18	8	4	2
129	8	17	13	18	4	7	9	12	19	16	5	10	20	15	11	3	2	1	14	6
130	4	11	6	19	8	5	7	10	13	2	20	18	9	12	15	17	14	1	16	3
131	17	3	20	2	13	19	18	5	9	16	7	12	8	10	11	14	4	15	1	6
132	1	18	2	10	19	14	3	4	20	12	5	17	16	8	7	15	6	13	9	11

133	3	8	1	10	12	20	4	15	11	9	13	14	5	7	16	19	2	6	18	17
134	12	1	5	14	16	15	13	9	6	3	19	8	17	11	18	2	7	20	10	4
135	10	5	18	20	15	3	13	16	4	6	9	19	14	17	1	12	7	2	11	8
136	1	13	14	16	7	9	6	11	10	12	8	2	17	15	5	3	4	18	19	20
137	17	16	7	3	2	5	14	15	4	8	18	9	11	20	12	6	10	19	13	1
138	8	19	6	5	10	12	20	1	17	18	2	4	13	3	11	7	16	14	15	9
139	5	6	13	1	18	11	2	16	3	8	4	19	17	10	12	7	14	20	9	15
140	6	18	4	16	20	3	17	12	13	9	14	2	8	5	10	19	7	1	11	15
141	18	8	2	3	6	14	9	20	13	4	11	10	12	19	5	15	1	7	16	17
142	20	8	3	18	12	10	9	15	5	11	19	14	4	17	7	13	2	1	16	6
143	13	2	4	18	8	14	1	15	6	17	11	9	5	10	12	20	7	3	19	16
144	14	7	1	3	5	10	20	11	12	4	9	18	13	17	2	19	15	16	6	8
145	12	7	13	17	16	4	1	2	18	14	8	9	3	15	10	5	6	20	11	19
146	13	8	15	20	9	1	18	4	16	3	12	2	5	19	10	14	6	11	7	17
147	16	11	4	18	6	20	1	5	14	7	3	10	13	19	9	15	8	17	2	12
148	6	20	13	8	9	10	16	2	14	11	18	7	3	4	17	5	19	1	15	12
149	19	8	1	3	14	15	2	13	9	6	11	12	5	18	17	4	7	10	16	20
150	7	18	9	4	2	12	10	15	1	11	8	17	19	5	16	13	6	14	3	20
151	3	14	20	15	4	1	16	11	6	2	5	12	10	18	7	19	9	17	8	13
152	4	12	1	6	17	2	9	7	5	14	18	16	11	10	13	8	15	19	20	3
153	16	5	11	7	18	14	20	1	13	2	17	6	4	9	12	15	8	10	19	3
154	5	20	10	1	12	19	15	16	7	2	11	14	3	8	13	9	4	17	6	18
155	14	6	18	2	1	10	16	3	8	9	11	13	7	15	5	20	4	17	12	19
156	7	12	10	14	9	2	11	1	3	19	20	8	17	16	4	6	13	5	15	18
157	3	5	2	10	12	17	4	7	13	9	8	6	11	16	20	19	18	15	1	14
158	3	10	8	4	9	6	5	2	13	14	11	1	20	17	15	16	19	7	18	12
159	3	5	19	4	6	12	20	17	18	16	7	11	10	15	13	8	14	1	2	9
160	20	15	2	19	12	17	14	5	7	4	9	6	16	11	18	13	3	10	1	8
161	11	14	2	15	5	1	4	7	3	10	17	8	16	18	20	19	9	13	6	12
162	10	6	9	16	15	7	19	1	2	18	3	8	11	5	17	4	13	20	12	14
163	14	15	19	6	8	5	1	9	11	7	17	2	4	10	13	3	12	16	18	20
164	3	1	11	14	20	17	6	9	13	15	12	5	19	16	4	10	18	2	8	7
165	13	19	14	16	11	20	6	12	1	2	17	18	3	4	9	10	7	8	15	5
166	10	2	15	16	7	17	14	6	18	5	4	9	19	11	3	1	20	13	8	12
167	2	4	7	19	3	8	14	5	9	20	10	13	11	15	1	18	17	6	12	16
168	5	6	20	10	14	18	4	3	7	9	1	12	15	2	8	13	19	16	11	17
169	13	2	8	3	16	10	17	4	15	19	12	18	11	7	6	5	14	20	1	9
170	9	2	16	18	15	6	11	3	4	5	7	14	20	17	13	8	19	12	10	1
171	16	15	1	14	19	8	10	6	20	5	3	12	13	9	7	4	18	2	17	11
172	16	9	18	3	11	20	15	19	1	4	5	12	17	10	13	14	2	8	6	7
173	15	3	9	18	2	11	7	12	5	1	20	4	6	14	10	16	19	13	8	17
174	3	11	8	20	7	16	18	12	13	2	6	1	10	15	14	17	19	4	9	5
175	7	8	9	4	12	19	13	14	3	16	15	17	10	5	18	6	11	1	20	2
176	7	8	1	12	13	5	4	10	9	3	18	14	15	19	2	11	20	17	16	6
177	18	11	8	14	13	9	10	2	7	20	3	19	12	15	4	1	6	16	5	17
178	9	2	20	19	5	17	14	16	12	15	13	4	6	18	1	11	10	3	7	8
179	16	3	6	10	2	14	4	19	9	8	20	17	1	13	18	11	12	15	5	7
180	16	8	4	10	12	5	11	18	6	1	9	14	20	15	3	2	13	17	7	19
181	2	13	3	9	8	14	7	20	4	1	19	5	18	15	6	12	17	16	11	10
182	1	13	12	2	16	17	8	6	15	18	19	7	4	3	11	10	5	20	9	14
183	14	6	9	1	18	13	10	20	4	19	8	7	2	16	5	12	11	15	17	3
184	13	4	12	15	2	3	1	16	20	11	8	18	6	10	9	14	17	5	19	7
185	19	7	13	17	16	11	12	4	14	3	1	8	10	6	15	5	20	2	9	18
186	4	19	14	13	17	2	6	7	3	18	16	12	20	11	9	1	5	8	15	10
187	20	10	6	14	19	5	9	1	15	13	16	8	3	4	7	12	17	2	11	18
188	19	10	1	12	18	17	14	20	6	15	5	11	3	2	16	8	4	9	7	13
189	3	19	14	10	9	5	17	20	7	12	6	15	13	18	1	8	16	4	2	11
190	20	16	7	2	8	15	1	14	11	12	3	5	6	17	4	18	13	19	9	10
191	7	11	18	8	2	20	1	3	10	5	6	14	4	16	15	17	9	13	19	12
192	9	19	6	2	10	16	3	4	15	1	17	8	7	11	5	14	13	12	18	20
193	2	15	5	7	4	10	17	3	12	8	6	1	20	9	11	18	16	19	14	13
194	11	1	2	18	7	5	19	20	16	13	10	4	3	12	6	8	9	17	15	14
195	12	18	20	4	9	1	5	10	11	2	16	13	6	19	3	15	14	17	8	7
196	3	5	20	8	9	15	17	4	7	13	12	6	14	19	2	10	11	16	18	1
197	10	13	15	19	6	20	2	12	16	3	17	5	14	9	18	4	8	11	1	7
198	20	12	19	4	7	9	1	10	16	8	2	11	14	6	13	18	5	15	3	17
199	19	17	4	1	8	20	14	10	7	11	12	13	6	3	16	2	18	9	5	15

2.3.3. Equipos resultantes

En la siguiente tabla, las columnas representan los equipos numerados del 0 al 19. Cada uno tiene el listado de los 10 jugadores resultantes de la

aplicación del algoritmo de Gale-Shapley.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
76	68	173	105	197	104	176	78	80	87	37	152	73	189	118	129	92	106	43	98
109	161	74	128	117	28	127	72	179	141	112	96	22	168	170	171	110	84	65	30
146	180	136	177	47	10	38	178	163	125	162	6	19	184	34	191	50	123	164	143
122	85	18	200	62	158	2	150	121	93	144	36	185	12	24	154	17	40	114	58
55	124	99	42	75	183	61	175	4	198	89	196	90	5	113	66	7	63	11	139
33	102	190	52	120	193	131	134	108	153	54	160	77	156	64	116	51	60	21	88
44	94	119	32	135	83	67	70	46	142	97	41	27	166	95	91	138	79	199	25
71	195	101	181	137	149	56	29	9	194	81	157	187	188	182	13	3	53	115	145
15	1	165	107	39	126	111	159	167	48	23	130	57	35	147	45	14	148	82	169
100	103	192	59	155	140	151	172	31	133	16	8	86	69	26	49	132	174	186	20

2.3.4. Instrucciones para la ejecución del algoritmo

Para poder ejecutar el algoritmo, es necesario tener la versión 2.7 de Python instalada. Si desea ejecutar el algoritmo con el set de pruebas provisto escriba la siguiente línea de comandos en la consola de su sistema operativo:

```
1 py -2 galeshapley.py
```

La ejecución de esta línea mostrará el contenido de los archivos de preferencias, y el resultado de la ejecución.