Facultad de Ingeniería
Universidad de Buenos Aires
Teoría de Algoritmos (75.29/95.06)

Trabajo Práctico N°1

2^{do} Cuatrimestre, 2018

	00000	
Boada, Ignacio	95212	
Geloso, Federico Pedro	98138	gelosofederico@gmail.com
Moreno, Fernando Ariel	96683	fer-ariel-94@hotmail.com

1. Parte 1: Variante de Gale Shapley

1.1. Explicación y pseudocódigo

Para la resolución del problema se utilizó como base el algoritmo de Gale Shapley visto en clase. En el algoritmo los recitales se proponen a las bandas, por lo tanto mientras hayan algún recital que no contrató a todas las bandas que puede y no se haya propuesto a todas bandas, el algoritmo tomará un recital que cumpla dicha condición y a la banda con mayor preferencia del recital a la que este no se haya propuesto todavía, se le propondrá. En caso que la banda tenga lugar para ir a ese recital, lo agregará a sus recitales, sino se debe analizar si entre los recitales de la banda hay alguno que tenga menor preferencia que el recital que se está considerando, en caso que así sea se extrae el recital de menor preferencia de los recitales de la banda y se agrega el nuevo recital, dejando un lugar libre para una nueva banda en el recital viejo, en caso contrario el recital deberá proponerse a la siguiente banda en su lista de preferencias hasta haberse propuesto a todas las bandas o haber contratado a la máxima cantidad de bandas que pueda contratar.

```
VARIANTE_GALE_SHAPLEY(recitales_preferencias, bandas_preferencias, X, Y):
    Mientras hayan recitales con lugares libres y no se hayan propuesto a todas las bandas:
        Elegir un recital que cumpla esas condiciones.
        Recorrer su lista de preferencias de manera decreciente, y para cada banda:
            Si el recital ya no puede contratar más bandas, pasar al siguiente recital.
            Si la banda puede tocar en un recital más:
                Agregar la banda al recital, y el recital a la banda.
            Sino:
                Mirar la lista de preferencias de la banda y si el recital actual está mejor
rankeado que alguno donde ya toca:
                    Cambiar el recital peor rankeado por el nuevo
    Imprimir los resultados mostrando las bandas que tocan en cada recital.
```

1.2. Análisis de complejidad

El algoritmo debe recorrer todos los recitales para asignar las bandas de acuerdo a sus preferencias, esto es O(N). Para cada recital, en el peor de los casos, debe recorrer todas las bandas, esto es O(M). Y para cada banda se debe analizar si el recital propuesto está mejor rankeado que los recitales que ya tiene, esto es O(Y^2), porque tiene que comparar los rankings de los recitales que ya tiene entre sí, obtener el peor rankeado y compararlo con el nuevo recital, para ver si lo agrega o no. Entonces en total tenemos que el algoritmo es O(N*M*Y^2).

1.3. Análisis de condiciones de retorno

Para que el algoritmo retorne un matching estable lo que tiene que suceder es que en dicho matching no tiene que haber ningún recital con una banda tal que exista otra banda que tenga mejor rankeado a ese recital que los recitales con los que matcheó y el recital mejor rankeada a esa banda. Para que el matching sea perfecto la cantidad de bandas tiene que ser igual a la de recitales, y la cantidad de bandas que se puede contratar debe ser igual a la cantidad de recitales a los que puede ir cada banda de manera que en el ranking de las bandas no se superponen preferencias (por ejemplo, si hay 3 y 3, sería (1,2,3), (2,3,1), (3,1,2)) , y ocurra lo mismo con los rankings de los recitales, y los rankings de los recitales se correspondan con los de las bandas, de manera que cada banda y recital se prefieren entre sí según un orden de preferencia.

1.4. Caso de preferencias similares

1.5. Análisis de casos particulares

A continuación se analizará el resultado del programa para 3 casos particulares. Los archivos que se utilizaron para la corrida se encuentran adjuntos a este informe en la versión digital (.zip)

- Caso 1: N = 10, M = 10, X = 1, Y = 1
- Caso 2: N = 10, M = 5, X = 2, Y = 2
- Caso 3: N = 10, M = 5, X = 2, Y = 1

Caso 1: Para este particular había 10 recitales y 10 bandas, y cada banda solo podía tocar en un recital y cada recital solo podía recibir una banda. Por esto el resultado final es un apareo de uno a uno de bandas y recitales, cumpliendo con el criterio de que los recitales reciben la máxima cantidad de bandas y las bandas tocan en el máximo de recitales posibles. Como casos específicos se pueden ver por ejemplo que el recital 1 queda apareado con la banda 4 y vemos que es correcto ya que ambos tienen como preferencia primera a esta pareja. También se ve que el recital 8 queda con la banda 3; para el recital esta es su primer preferencia, pero para la banda es su tercera, pero los dos recitales que preceden terminan apareados con su primer preferencia, que no es la banda 3. Por lo tanto no hay inestabilidad.

Caso 2: Aquí había 10 recitales, 5 bandas y la cantidad de bandas por recital y el número de recitales donde podía tocar cada banda era de 2. Por lo tanto se puede ver que todas las bandas tocan en dos recitales, pero sin embargo en el caso de los recitales, hay algunos con 2, otros con 1 y otros con 0 bandas. Por ejemplo, el recital 1 queda con las bandas 2 y 3. Para estas bandas, la preferencia de este recital y el otro donde quedan apareados es más alta que las de los recitales 4,7 y 8 y por esto sumado a las preferencias de las demás bandas, esos tres recitales terminan vacios.

Caso 3: Este caso es como el anterior, pero la cantidad de recitales donde puede tocar cada banda se reduce a uno. El resultado es prácticamente idéntico al anterior, y solo se quita una aparición de cada una de las bandas. Así nuevamente quedan recitales con 2, 1 y 0 bandas.

Nota: El programa se ejecuta pasándole por parámetro los números N, M, X, Y. De forma predeterminada para cada corrida se generan los archivos necesarios de forma aleatoria. Si se quiere utilizar algunos archivos en particular se debe quitar la línea 64 dentro del archivo Variante_Gale_Shapley.py, dentro de la función main, que dice "manejo_archivos.generar_archivos(N, M)".

1.6. Comparación de complejidad teórica con la programada

2. Parte 2: Complejidad algorítmica

Para esta sección se consideró el problema de calcular todos los números primos menor que un valor N. Se consideraron 2 algoritmos para resolverlo: un algoritmo por fuerza bruta, que prueba la divisibilidad con todos los números menores, y el algoritmo de la criba de Erastótenes.

El primer algoritmo es fácil de ver y analizar. En pseudocódigo:

```
PrimosNaive(N):
    Sea L una lista.
    EsPrimo := Verdadero.
    Para cada i entre 2 y N:
        j = 2.
        Mientras j < i y EsPrimo = Verdadero:
            Si i es divisible por j:
                EsPrimo := Falso.
            j = j + 1.
        Si EsPrimo = Verdadero:
            Agregar i al final de L.
    Devolver L
```

En esto se puede ver que L contendrá todos los números que pasaron el chequeo. La complejidad temporal del algoritmo se puede ver que es $\mathcal{O}(n^2)$, donde n es el número de entrada. Es así ya que itera por cada número todos los números menores que este, teniendo entonces $T(n) = \sum_{i=2}^{N} \sum_{j=2}^{i} 1 = \sum_{i=2}^{N} i - 2 = (N-1)(N-2)/2$.

Este algoritmo se podría mejorar, pero se va a analizar el otro algoritmos propuesto. En pseudocódigo:

```
PrimosCriba(N):
    Sea L una lista de booleanos de 0 a N-1, seteados en Verdadero.
    Para cada i de 2 a ⌊√N⌋:
        Si L[i] = Verdadero:
            Para cada j de i^2 a N de a pasos de a i:
                L[j] := Falso
    Sea P la lista de números i tal que L[i] = Verdadero
    Devolver P
```

Se puede ver que para cada primo p menor que $\sqrt{N}$, hace $\frac{N}{p}$ operaciones. A partir del segundo teorema de Mertens, se tiene que:

$$\lim_{n\to\infty} \sum_{p<x} \frac{1}{p} = ln\ ln\ n + M \tag{1}$$

Siendo p los números primos y M una constante que a efectos del orden asintótico no es relevante. La optimización de hacer hasta $\sqrt{N}$ no es relevante en el orden asintótico tampoco, ya que estaría dentro del logaritmo y sale multiplicando del primer logaritmo y sumando del segundo. Con esto, queda que $N \sum_{p<\sqrt{N}} \frac{1}{p} \in \mathcal{O}(N \cdot log\ log\ N)$.

Todo esto asume que el acceso a la lista L es $\mathcal{O}(1)$. Al programarlo se utilizó la estructura *list* de Python, que asegura el acceso en tiempo constante. Dado que se pide devolver una lista con los números, al final se recorre esta estructura y se agregan a la lista final a devolver todos los que terminaron siendo Verdaderos, lo cual tiene un coste lineal. A su vez se tuvo en cuenta una optimización más, que es que los primos i después del 2 se pueden recorrer desde i^2 a N en pasos de a $2i$, ya que se sabe que i es impar (el único primo par es 2) y entonces $i^2 + n \cdot i$ es par si n es impar, lo cual ya se sabe que no es primo. Por esto se separaron el caso del 2 del resto.

2.1. Documentación del programa

Para ejecutar el programa se deben pasar por línea de comandos el número N, el método como E (Erastótenes) o F (fuerza bruta), y si se quieren imprimir los datos, se pasa -T para escribir el tiempo, y -L para la lista. Esto se escribirá en la salida estándar, por lo que para escribir en un archivo se puede redirigir el flujo de salida.

2.2. Corridas de prueba

Se corrió el programa varias veces con valores de N múltiplos de 10, para ambos casos. En la siguiente figura se pueden ver graficadas con ambas escalas logarítmicas.

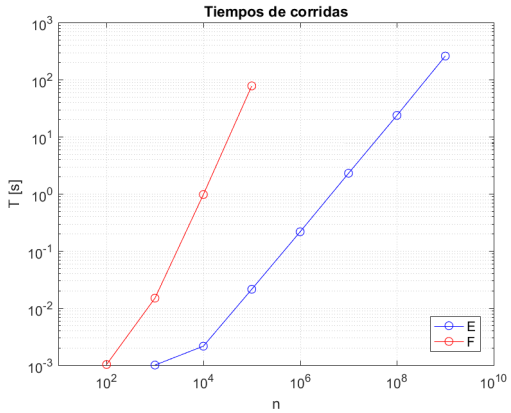

Figura 1: Tiempos de ejecución

Al ser logarítmico en ambos ejes, ambas funciones polinomiales se verán como rectas, pero se puede ver claramente que las pendientes son distintas, ya que uno es $\mathcal{O}(n^2)$ mientras que el otro es $\mathcal{O}(n)$ (el loglog se descarta ya que no se puede notar a menos que sean números muy grandes). La forma más simple de comprobar eso es viendo los datos usados, a continuación:

n	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
E		$1{,}00 \cdot 10^{-3}$	$2{,}14 \cdot 10^{-3}$	$2{,}13 \cdot 10^{-2}$	0.216	2.29	23.6	259
F	$1{,}02 \cdot 10^{-3}$	$1{,}49 \cdot 10^{-2}$	0.971	77.5				

En la tabla se pueden ver que los tiempos de el método E aumentan en 10 cuando la entrada aumenta en 10. En el método F, se ve principalmente en el salto de $n = 10^4$ a $n = 10^5$ que se multiplica casi 100 veces ($77{,}5/0.971 = 80$).

Todos los casos con tiempos menores al minuto se obtuvieron con un promedio de 10 corridas. EN el tiempo no se incluye el tiempo que tarda en imprimir, solo en calcular. Cantidades mayores para el algoritmo de fuerza bruta implicarían un tiempo de 6000 segundos, aproximadamente hora y media, por lo que se consideró innecesario. En el caso de la criba, tiempos mayores implicarían mucho más cantidad de memoria (es lineal en la memoria), por lo que se verían 10^{10} enteros en memoria, lo cual, si son enteros de 4 bytes, son $4 \cdot 10^{10}$ bytes en memoria, que son 40 gigabytes.

3. Códigos

Listing 1: Variante_Gale_Shapley.py

```
1  import sys, manejo_archivos
2
3  #N = Cantidad de recitales
4  #M = Cantidad de bandas
5  #X = Cantidad de bandas que pueden contratar los recitales
6  #Y = Cantidad de recitales en los que puede participar una banda.
7
8  def variante_Gale_Shapley(recitales_preferencias, bandas_preferencias, X, Y):
9          recitales_libres = []
10         bandas_por_recital = {}
11         recitales_por_banda = {}
12         for i in range(1, N+1): #O(N)
13                 recitales_libres.append(i)
14                 bandas_por_recital[i] = []
15         for j in range(1, M+1):  #O(M)
16                 recitales_por_banda[j] = []
17         recitales_ult_propuesto = [0] * N
18
19         while len(recitales_libres)>0: #O(N)
20                 recital_act = recitales_libres.pop() #O(1)
21                 for i in range(recitales_ult_propuesto[recital_act-1], M): #O(M)
22                         banda_act = recitales_preferencias[recital_act][i]
23                         if len(bandas_por_recital[recital_act]) == X:
24                                 break
25                         if len(recitales_por_banda[banda_act]) < Y:
26                                 recitales_por_banda[banda_act].append(
                                       recital_act)
27                                 bandas_por_recital[recital_act].append(banda_act
                                       )
28                         elif banda_prefiere_recital_nuevo(banda_act, recital_act
                               , recitales_por_banda, bandas_preferencias): #O(Y^2)
29                                 recital_viejo =
                                       reemplazar_recital_viejo_por_nuevo(banda_act,
                                        recital_act, recitales_por_banda,
                                       bandas_por_recital, bandas_preferencias)#O(Y
                                       ^2)
30                                 if recitales_ult_propuesto[recital_viejo-1] < M:
31                                         recitales_libres.append(recital_viejo)
32                         recitales_ult_propuesto[recital_act-1] += 1
33
34         imprimir_resultados(bandas_por_recital)
35
36
37 def banda_prefiere_recital_nuevo(banda_act, recital_act, recitales_por_banda,
       bandas_preferencias):  #O(Y^2)
```

```
38          for x in range(len(bandas_preferencias[banda_act])-1, 0, -1): #O(Y)
39                  if bandas_preferencias[banda_act][x] in recitales_por_banda[
                        banda_act]: #O(Y)
40                          return True
41                  if bandas_preferencias[banda_act][x] == recital_act:
42                          return False
43
44
45  def reemplazar_recital_viejo_por_nuevo(banda_act, recital_act,
        recitales_por_banda, bandas_por_recital, bandas_preferencias):#O(Y^2)
46          for x in range(len(bandas_preferencias[banda_act])-1, 0, -1): #O(Y)
47                  if bandas_preferencias[banda_act][x] in recitales_por_banda[
                        banda_act]: #O(Y)
48                          recital_viejo = bandas_preferencias[banda_act][x]
49                          recitales_por_banda[banda_act].remove(recital_viejo)
50                          recitales_por_banda[banda_act].append(recital_act)
51                          bandas_por_recital[recital_viejo].remove(banda_act)
52                          bandas_por_recital[recital_act].append(banda_act)
53                          return recital_viejo
54
55
56  def imprimir_resultados(bandas_por_recital):
57          print("Recital | Bandas que tocan")
58          print("-------------------------------")
59          for x in range(1, len(bandas_por_recital)+1):
60                  print("   {}     | {}".format(x, bandas_por_recital[x]))
61
62
63  def main(N,M,X,Y):
64      manejo_archivos.generar_archivos(N, M)
65      recitales_preferencias, bandas_preferencias = manejo_archivos.leer_archivos(
            N, M)
66      variante_Gale_Shapley(recitales_preferencias, bandas_preferencias, X, Y)
67      print("Fin del programa")
68
69
70
71  if len(sys.argv)!=5:
72          print("Se deben ingresar los cuatro parametros: N, M, X e Y")
73          exit()
74  try:
75      N = int(sys.argv[1])
76      M = int(sys.argv[2])
77      X = int(sys.argv[3])
78      Y = int(sys.argv[4])
79  except ValueError:
80          print("Se deben ingresar 4 valores numericos")
81          exit()
82
83  main(N,M,X,Y)
```

Listing 2: manejo_archivos.py

```
1  import random
2
3  #N = Cantidad de recitales
4  #M = Cantidad de bandas
```

```
5  #X = Cantidad de bandas que pueden contratar los recitales
6  #Y = Cantidad de recitales en los que puede participar una banda.
7
8  #Genera N archivos de recital con sus preferencias en forma random, y M archivos
       para las bandas con sus preferencias random.
9  def generar_archivos(N, M):
10     for i in range(N):
11         recital_nro = 'recital_' +  str(i+1) + '.txt'
12         with open(recital_nro, 'w+') as recital:
13             recital_preferencias = random.sample(range(1, M + 1), M)
14             for preferencia in recital_preferencias:
15                 recital.write(str(preferencia))
16                 recital.write(str('\n'))
17     for i in range(M):
18         banda_nro = 'banda_' +  str(i+1) + '.txt'
19         with open(banda_nro, 'w+') as banda:
20             banda_preferencias = random.sample(range(1, N + 1), N)
21             for preferencia in banda_preferencias:
22                 banda.write(str(preferencia))
23                 banda.write(str('\n'))
24
25 def leer_archivos(N, M):
26     recitales_preferencias = {}
27     for i in range(1, N + 1):
28         recital_nro = 'recital_' +  str(i) + '.txt'
29         recital_preferencias = []
30         with open(recital_nro, 'r') as recital:
31             for preferencia in range(M):
32                 recital_preferencias.append(int(recital.readline()))
33         recitales_preferencias[i] = recital_preferencias
34     bandas_preferencias = {}
35     for i in range(1, M + 1):
36         banda_nro = 'banda_' +  str(i) + '.txt'
37         banda_preferencias = []
38         with open(banda_nro, 'r') as banda:
39             for preferencia in range(N):
40                 banda_preferencias.append(int(banda.readline()))
41         bandas_preferencias[i] = banda_preferencias
42     return recitales_preferencias, bandas_preferencias
```

Listing 3: PrimeNumbers.py

```
1  # -*- coding: utf-8 -*-
2  # TP1 TDA Parte 2: Complejidad algortmica
3  # Obtener una lista de primos hasta el nmero N.
4  # Se puede hacer de dos mtodos, con la criba de Erasttenes (llamando con E) o
5  # por fuerza bruta (llamando con F)
6
7  import time
8  import argparse
9
10 def main(N,M, time_flag, list_flag):
11     ## Lectura de command line
12     #N = 100
13     start_time = time.time()
14     if M == "E":
15         prime_list = primes_until_N_E(N)
```

```
16      if M == "F":
17          prime_list = primes_until_N_F(N)
18      diff = time.time()-start_time
19      if list_flag == True:
20          print(prime_list)
21      if time_flag == True:
22          print(diff)
23
24
25  def primes_until_N_E( N ):
26      # Devuelve una lista L con los primos hasta N.
27      # Implementa la criba de Erasttenes
28      if N == 1:
29          return []
30      if N == 2:
31          return []
32      primes = [True] * N;
33      primes[0] = primes[1] = False;
34      # Saco los pares para mejorar despus la criba
35      for i in range(4,N,2):
36          primes[i] = False
37      # Recorro los nmeros y acto para los impares, empezando desde el nmero al
38      # cuadrado y movindome de a 2 en sus mltiplos, porque ya se que i*i es
39      # impar, impar+impar es par y esos ya los descart antes
40      for i in range(1,N,2):
41          if primes[i] == True:
42              for j in range(i*i,N,2*i):
43                  primes[j] = False
44      # Ahora formo la lista con los que quedaron
45      final_primes = [2]
46      for i in range(3,N,2):
47          if primes[i] == True:
48              final_primes.append(i)
49      return final_primes
50
51  def primes_until_N_F( N ):
52      # Devuelve una lista L con los primos hasta N.
53      # Lo hace de forma Naive
54      if N == 1:
55          return []
56      primes = []
57      for i in range(2,N):
58          isprime = True
59          j = 2
60          while isprime == True and j < i:
61              if i%j==0:
62                  isprime = False
63              j = j+1
64          if isprime == True:
65              primes.append(i)
66      return primes
67
68
69
70  if __name__ == '__main__':
71      parser = argparse.ArgumentParser(description='parser.')
```

```
72      parser.add_argument('number',type=int)
73      parser.add_argument('method',choices=['E', 'F'])
74      parser.add_argument('-T', '--time', action='store_true')
75      parser.add_argument('-L', '--list', action='store_true')
76      parsed_args = parser.parse_args()
77      N = parsed_args.number
78      M = parsed_args.method
79      time_flag = parsed_args.time
80      list_flag = parsed_args.list
81      main(N,M,time_flag,list_flag)
```