Estructuras de Datos y Algoritmos: 2022-1

Miguel Arturo Reina Rocabado, 202014739, m.reina@uniandes.edu.co

Sergio Andrés Oliveros Barbosa, 202123159, s.oliverosb@uniandes.edu.co

Los requerimientos 1, 3 y 5 se realizaron en la máquina 1, mientras que los requerimientos 2, 4 y 6 se realizaron en la máquina 2.

	Máquina 1	Máquina 2
Procesadores	AMD Ryzen 7 5800H 3.20 GHz	AMD Ryzen 5 3500U 2.10Ghz
Memoria RAM (GB)	8.00 GB, 7.35GB usable	8.00GB, 5.95 usable
Sistema Operativo	Windows 11 - 64bits	Windows 11 – 64 bits

Análisis de resultados

Requerimiento 1

Entradas: `Nombre del club`

Salidas: `Número total de adquisiones del club`

`league_level, league_name`

`Cinco jugadores más recientes vinculados al club`

Análisis de complejidad:

Precarga: en esta parte creamos un map que tenga como llaves el nombre del club y como valor un ordered_map y agrega el jugador dentro de este om.

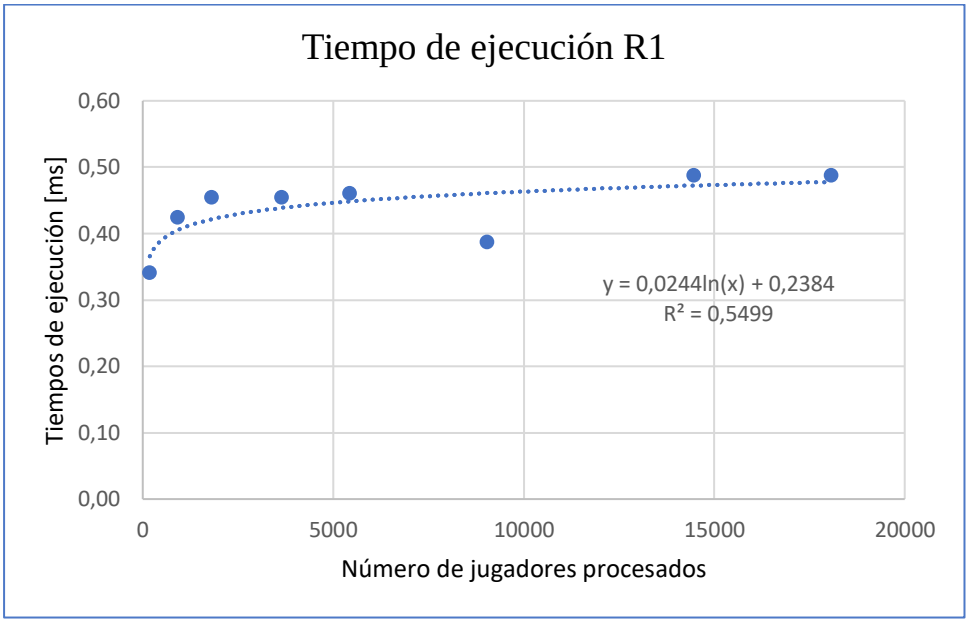
```
catalog['R1_by_club'] = mp.newMap(numelements=701,
                                  maptype='PROBING',
                                  loadfactor=our_loadfactor,
                                  prime=our_prime)
```

```
def R1_map_by_club(catalog, player):
    club_name = player['club_name']


    exist_club = mp.get(catalog['R1_by_club'], club_name)
    if (exist_club is None):
        new_om = om.newMap(omaptype='RBT', comparefunction= cmp_r1)
        mp.put(catalog['R1_by_club'], club_name, new_om)

    ordered_map = mp.get(catalog['R1_by_club'], club_name)['value']
    llave_compuesta = (player['club_joined'], player['age'], player['dob'], player['short_name'])
    om.put(ordered_map, llave_compuesta, player)
```

Para agregarlo al ordered_map, se utiliza una llave compuesta que tiene el criterio de ordenamiento pedido por el requerimiento (fecha de unión, edad, fecha de nacimiento y nombre corto, respectivamente) y como valor tendrá el jugador.

Con respecto al tiempo de ejecución del requerimiento, encontramos los siguientes datos, resumidos en la siguiente gráfica:

Tamaño del archivo	Número de jugadores	Tiempo de ejecución [ms]	Consumo de memoria [kB]
small	184	0.34	5.8
5pct	906	0.42	5.8
10pct	1813	0.45	5.8
20pct	3637	0.45	6.36
30pct	5427	0.46	6.89
50pct	9039	0.39	6.83
80pct	14467	0.49	7.39
large	18076	0.49	7.89



Aquí claramente la mayoría de datos (exceptuando 50pct) poseen una tendencia logarítmica. Esto se explica cuando analizamos el código una vez el usuario ingresa 1:

```
def r1_answer(catalog, inp_club_name):

    exist_club = mp.get(catalog['R1_by_club'], inp_club_name)  #O(1)
    #Contención de error
    if (exist_club is None):
        return None, None, None, None, None, None

    ordered_map = exist_club['value']                          #O(1)
    n_adquisitions = om.size(ordered_map)                      #O(1)

    player_list = tomar_5_ultimos(ordered_map, n_adquisitions) #O(log(n))
    player_x = lt.getElement(player_list, 1)                   #O(1)
    league_name = player_x['league_name']                      #O(1)
    league_level = player_x['league_level']                    #O(1)

    #Lab9
    height = indexHeight(ordered_map)                          #O(1)
    n_elements = indexSize(ordered_map)                        #O(1)



    return n_adquisitions, player_list, league_name, league_level, height, n_elements
```

Las primeras operaciones con de tiempo constante, pues hacer mp.get es O(1), encontrar el tamaño de un mapa ordenado es O(1), conseguir el primer elemento, un parámetro de un jugador, la altura y el número de elementos son todos de tiempo constante. Sin embargo, para poder tomar los primeros 5 (los 5 últimos jugadores ingresados al club) se necesita pasar por la altura del ordered map, este como es un RBT devolver los primeros 5 será O(altura) que por ser RBT O(log(n)). Al sumar las complejidades, nos damos cuenta que esta última operación es la más costosa y por eso la gráfica anterior posee un comportamiento logarítmico. Por último, explicamos el outlier de 50pct por la conformación del RBT que pudo favorecer los 5 primeros jugadores a una menor que sus tamaños de muestra cercanos.

Requerimiento 2

Entradas:

`Posición, Límite inferior para el desempeño global, Límite superior para el desempeño global, Límite inferior para el potencial, Límite superior para el potencial, Límite inferior para el salario, Límite superior para el potencial.`

Salidas:

`Lista con los primeros y últimos 3 jugadores que cumplen los criterios, Número total de jugadores en una posición.`

Precarga:

```
catalog["R2_position_trees"] = mp.newMap(numelements=20,
                                         maptype='PROBING',
                                         loadfactor=our_loadfactor,
                                         prime=our_prime,
                                         comparefunction=comparePositions)
```

Inicialmente en el catálogo se crea una Tabla de Hash con estructura de datos Linear Probing. En ella se almacenarán parejas llave, valor, donde las llaves serán las posibles posiciones de juego y los valores serán Árboles Rojo-Negro que contendrán a todos los jugadores que juegan dentro de esa posición, organizados por una llave compuesta de (overall, potential, wage, age name).

```
def R2_RBT_for_pos(catalog, player):
    pos_list = player["player_positions"].split(",")
    for pos in pos_list:
        pos = pos.strip(" ")

        exist_pos = mp.get(catalog["R2_position_trees"], pos)

        if exist_pos is None:
            new_rbt =  om.newMap(omaptype="RBT", comparefunction= cmp_r2)
            mp.put(catalog["R2_position_trees"], pos, new_rbt)

        ordered_map = mp.get(catalog["R2_position_trees"], pos)['value']
        llave_compuesta = (player["overall"], player["potential"], player["wage_eur"], player['age'], player["short_name"])
        om.put(ordered_map, llave_compuesta, player)
```

En la función de carga específicamente se obtienen las posiciones donde el jugador puede jugar para luego agregar al jugador en los diferentes árboles de posiciones con la llave compuesta descrita anteriormente.

Análisis de complejidad:

```python
def R2_answer(catalog, pos_player, min_overall, max_overall, min_potential, max_potential, min_wage, max_wage):
    exist_map = mp.get(catalog["R2_position_trees"],pos_player)

    if exist_map == None:
        return None, None

    RBT = exist_map["value"]

    low_key = (min_overall, min_potential,  min_wage,  0,"")
    high_key = ( max_overall,  max_potential, max_wage, 200,"zzzzzzzzzzzzzzz")

    keys_in_range = om.keys(RBT,low_key, high_key)
    keys_in_range = correct_range(keys_in_range, min_potential, max_potential, min_wage, max_wage)

    list_jugadores_keys = lt.newList("ARRAY_LIST")

    num_jugadores = lt.size(keys_in_range)

    for i in range(1,7):
        if i < 4:
            jugador = lt.getElement(keys_in_range, i)
            lt.addFirst(list_jugadores_keys, jugador)
        else:
            jugador = lt.getElement(keys_in_range, num_jugadores - (6-i))
            lt.addFirst(list_jugadores_keys, jugador)

    list_jugadores = lt.newList("ARRAY_LIST")

    for key in lt.iterator(list_jugadores_keys):
        jugador = om.get(RBT, key)["value"]
        lt.addLast(list_jugadores, jugador)

    return list_jugadores, num_jugadores
```

1) El proceso para obtener la respuesta deseada comienza por verificar que exista un Árbol asociado a la llave que ingrese el usuario. Acción con un costo **O(1)**.
2) Una vez con el árbol, se utilizan los valores ingresados por el usuario para crear 2 llaves compuestas que determinaran el rango de los valores que se desean obtener. La operación para obtener estos valores mediante el uso de la función *om.keys()* es **O(altura + # numelements)** dónde numelements es la cantidad de jugadores que cumplen el primer rango.

```python
def correct_range(keys, min_potential, max_potential, min_wage, max_wage):

    list_good_elems = lt.newList("ARRAY_LIST")
    for key in lt.iterator(keys):
        overall, potential,  wage,  age, name = key

        if potential in range(min_potential, max_potential+1):
            if wage in range(min_wage, max_wage+1):
                lt.addLast(list_good_elems, key)

    return list_good_elems
```

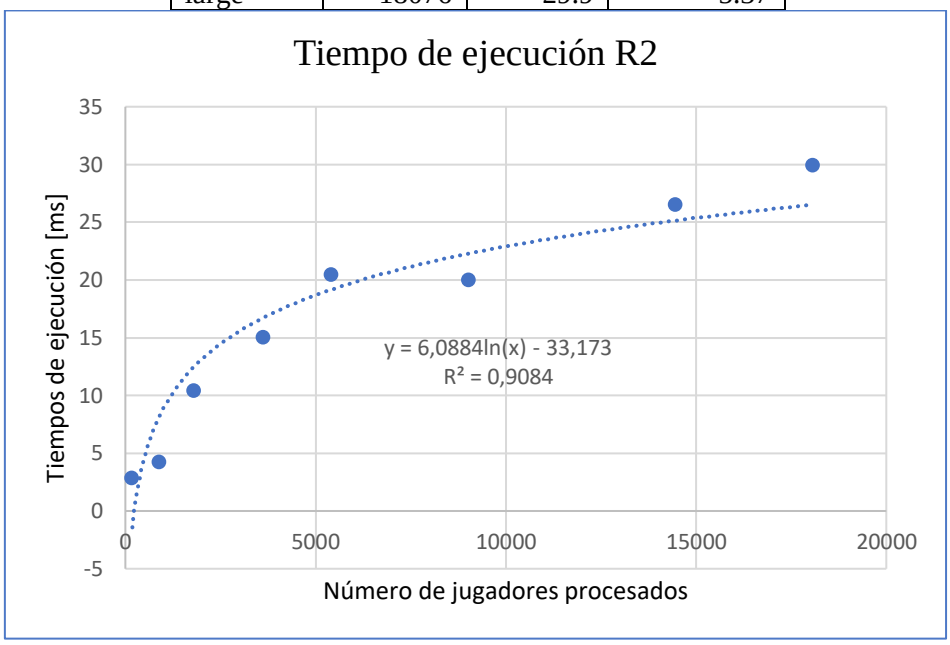
3) Una vez con los valores, sucede que la función *om.keys()* incluye a todos los elementos que se encuentren dentro del rango de los primeros componentes de la llave compuesta. En otras palabras, se obtienen todas las llaves dentro del rango de "overall". Por lo que para descartar aquellos elementos que no deberían tenerse en cuenta debido a que no se encuentran dentro de los otros rangos ingresados por el usuario, se recorre la lista de llaves para encontrar los elementos que cumplen con los parámetros ingresados, luego se guardan en una lista y una vez se hayan recorrido todas las llaves se retorna la lista con los elementos que cumplen con las indicaciones del usuario. Esta operación tiene un costo de **O(# numelements)**.
4) De la anterior lista se obtiene las primeras 3 y últimas 3 llaves, las cuales sabemos que están en el orden adecuado debido a la precarga, y luego se usan las llaves para encontrar los diccionarios asociados y retornas los 6 diccionarios. Acciones que tendrían un costo total de **O(12)**.

Dando una complejidad final de **O(1 + altura + #numelements + # numelements + 12)**.

La cual se puede simplificar a **O(altura + 2 #numelements)**, y dejando el termino más grande (altura es logarítmica a comparación de n datos pues es un RBT) quedaría en **O(#numelements)**. La cual se puede representar de la forma O(n). Ahora, ya depende de la distribución de los datos en el rango escojido cuál es la relación entre numelements y n.

Gráfica de tiempos de ejecución:

Tamaño del archivo	Número de jugadores	Tiempo de ejecución [ms]	Consumo de memoria [kB]
small	184	2.85	43.54
5pct	906	4.23	60.18
10pct	1813	10.38	61.21
20pct	3637	15.02	61.3
30pct	5427	20.44	61.78
50pct	9039	19.95	62.47
80pct	14467	26.47	7.99
large	18076	29.9	5.37



Al realizar la gráfica del tiempo del requerimiento contra el número de jugadores en el archivo que cargamos, vemos que esta presenta un comportamiento logarítmico, hecho que se puede verificar al ver que el R^2 de la gráfica es mayor cuando se hace una regresión logarítmica a cuando se hace una lineal. La razón de esto es que nuestro algoritmo lineal no se ejecuta sobre el número total de elementos, sino sobre una fracción de estos. En el requerimiento se nos pedían todos los jugadores de una posición especifica que cumplieran ciertos criterios. Sin embargo, al solicitar los jugadores de una posición particular se es posible descartar a una gran parte del total de los datos. Este solo se presentaría su a medida que crece el número de jugadores, menor es el crecimiento que tienen los jugadores en una posición especifica. Por lo que esto da cuenta de que los jugadores están distribuidos desigualmente, y que justamente la posición sobre la cual decidimos realizar las pruebas es una con poco crecimiento.

Entradas:

Límite inferior del salario recibido por los jugadores (wage_eur).

Límite superior del salario recibido por los jugadores (wage_eur).

Una de las características que identifican a los jugadores (player_tags).

Salidas:

número de jugadores que cumplen con los criterios

primeros 3 y últimos 3 jugadores con esa característica y rango salarial

Precarga:

```
def R3_by_tag_wage(catalog, player):
    list_tags = player['player_tags'].split(',')
    for tag in list_tags:

        if tag == '':
            pass
        else:
            tag = tag.strip(' ')
            exist_tag = mp.get(catalog['R3_by_tag_wage'], tag)

            if (exist_tag is None):
                new_om = om.newMap(omaptype='RBT', comparefunction=cmp_r3)
                mp.put(catalog['R3_by_tag_wage'], tag, new_om)

            ordered_map = mp.get(catalog['R3_by_tag_wage'], tag)['value']
            #llave con 4 criterios
            llave_compuesta = (player['wage_eur'], player['overall'], player['potential'], player['long_name'])
            om.put(ordered_map, llave_compuesta, player)
```

Para la precarga, extraemos todos los *tags* asociados a un jugador y creamos una Tabla de Hash en la que estas sean las llaves y los valores serán *ordered maps* que contengan los jugadores de manera ordenada dado los 4 criterios del requerimiento.

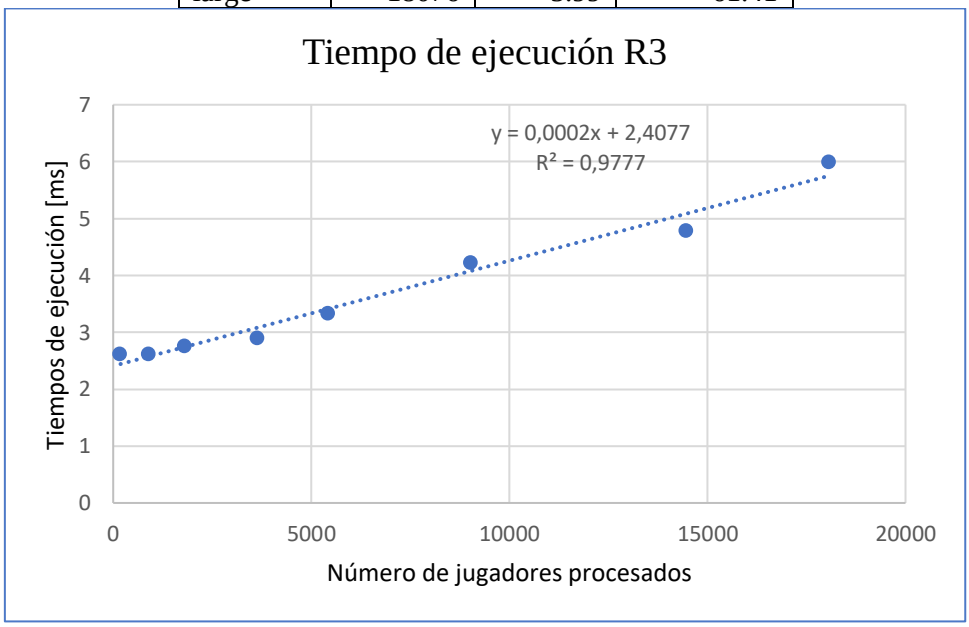
Análisis de complejidad:

Una vez el usuario ingresa los parámetros, se busca la llave dentro de la Tabla de Hash creada, se ingresa al valor, se crea una lista en ED de arreglo y se crean las llaves. Todo esto es en tiempo constante O(1), pues no dependen de la cantidad de datos tomados. Por el contrario, al momento de realizar *om.keys* la complejidad es de O(altura + numelementos). Donde numelements corresponde a la cantidad de jugadores que cumplen el tag dado. Posteriormente se toma el tamaño de la TAD lista resultante y se toman los primeros 3 y los últimos 3 elementos de la lista.

```python
#-----------------------R3---------------------------------
def r3_answer(catalog, in_wage_lo, in_wage_hi, in_tag):
    exist_tag = mp.get(catalog['R3_by_tag_wage'], in_tag)   #O(1)
    #Contención error
    if exist_tag is None:
        return None, None

    ordered_map = exist_tag['value']                    #O(1)

    rta_list = lt.newList('ARRAY_LIST')                 #O(1)

    low_key = (in_wage_hi, 99999999999, 9999999999, "zzzzzzzzzzzzzzz")  #O(1)
    high_key = (in_wage_lo, 0, 0, '')                   #O(1)

    keys_in_range = om.keys(ordered_map, low_key, high_key) #O(altura + numelementos)
    #out
    n_elements = lt.size(keys_in_range)  #O(1)

    rta_list = get3first_3last(keys_in_range, ordered_map)  #O(1)

    return n_elements, rta_list
```

Gráfica de tiempos de ejecución:

Tamaño del archivo	Número de jugadores	Tiempo de ejecución [ms]	Consumo de memoria [kB]
small	184	2.61	39.97
5pct	906	2.61	42.69
10pct	1813	2.75	47
20pct	3637	2.89	57.36
30pct	5427	3.33	59.83
50pct	9039	4.21	59.74
80pct	14467	4.78	60.48
large	18076	5.99	61.41



A través de esta gráfica evidenciamos que la complejidad es directamente proporcional con la cantidad de elementos a analizar O(n). Esto puede ser explicado si tenemos la hipótesis que la

cantidad de jugadores que cumplen el *tag* dado es directamente proporcional a la cantidad de jugadores a analizar, por lo que la operación más costosa que sería O(altura + numlements) puede ser expresada como O(log(n) + n) pues es un RBT y por operaciones de notación O eso sería igual a O(n).

Requerimiento 4 – Sergio Oliveros::202123159

Entradas:

`Límite inferior de la fecha de nacimiento, Límite superior de la fecha de nacimiento, Característica de los jugadores.`

Salidas:

`Lista con los primeros y últimos 3 jugadores que cumplen con los criterios, número de jugadores que comparten la misma característica.`

Precarga:

```
catalog["R4_trait_trees"] = mp.newMap(numelements=30,
                                      maptype='PROBING',
                                      loadfactor=our_loadfactor,
                                      prime=our_prime,
                                      comparefunction= compareTraits)
```

Se comienza por crear una Tabla de Hash implementada con Linear Probing en el catálogo, organizada de tal manera que se tiene de llaves, los rasgos de los jugadores, y de valores, Árboles Rojo Negro donde los jugadores están organizados de acuerdo con una llave compuesta que contiene la fecha de nacimiento, el desempeño general, el potencial y el nombre del jugador.

```
def R4_RBT_for_trait(catalog, player):
    if player["player_traits"] == "":
        pass
    else:
        list_traits = player["player_traits"].split(",")
        for trait in list_traits:
            trait = trait.strip(" ")

            exist_trait = mp.get(catalog["R4_trait_trees"], trait)

            if exist_trait is None:
                new_rbt =  om.newMap(omaptype="RBT", comparefunction= cmp_r4)
                mp.put(catalog["R4_trait_trees"], trait, new_rbt)

            ordered_map = mp.get(catalog["R4_trait_trees"], trait)['value']

            llave = (player["dob"], float(player["overall"]), float(player["potential"]), player["long_name"])
            om.put(ordered_map, llave, player)
```

Ahora, haciendo referencia a la función de carga que se utiliza en el requerimiento 4, vemos que esta lo que hace es verificar que un jugador posea comentarios, y en caso de que los tenga, ubica al jugador dentro de todos los Árboles Rojo Negro a los que debería pertenecer según sus comentarios.

Análisis de complejidad:

```python
def R4_answer(catalog, trait, min_dob, max_dob):

    min_dob = str_to_date(min_dob)
    max_dob = str_to_date(max_dob)

    exists_trait = mp.get(catalog["R4_trait_trees"], trait)

    if exists_trait is None:
        return None, None

    trait_RBT = exists_trait["value"]

    keylo = (min_dob,0, 0, "")
    keyhi = (max_dob, 0, 0, "" )

    keys_in_range = om.keys(trait_RBT, keylo, keyhi)

    list_jugadores = lt.newList("ARRAY_LIST")

    for key in lt.iterator(keys_in_range):
        jugador = om.get(trait_RBT, key)["value"]
        lt.addLast(list_jugadores, jugador)

    resp_list = lt.newList("ARRAY_LIST")
    num_jugadores = lt.size(list_jugadores)
    for i in range(1,7):
        if i < 4:
            jugador = lt.getElement(list_jugadores, i)
            lt.addFirst(resp_list, jugador)
        else:
            jugador = lt.getElement(list_jugadores, num_jugadores - (6-i))
            lt.addFirst(resp_list, jugador)

    return resp_list, num_jugadores
```

1) Se convierten las fechas ingresadas por el usuario en objetos *datetime* para poder realizar las comparaciones de manera apropiada. Esto tiene una complejidad de O(1).
2) Se verifica que exista un Árbol Rojo Negro para el comentario ingresado. O(1).
3) Se establece el rango de llaves para el cual se desean elementos y se hace *om.keys()* para dichos elementos. La complejidad de esta operación es O(Altura + # elementos del árbol).
4) Debido a que los elementos dentro del rango son retornados en una lista enlazada, tiene una menor complejidad recorrer todos los elementos una vez para crear un arreglo y luego obtener los primeros 3 y últimos 3 jugadores de la lista. Comparado a simplemente solicitar los 6 elementos deseados, pues para obtener los últimos elementos se tendría que recorrer aproximadamente todos los elementos de la lista varias veces. Por lo que es más fácil hacer un único recorrido que tendría una complejidad de **O(# de llaves en el rango)**.
5) Finalmente, y como se mencionó anteriormente, se obtienen los 6 elementos deseados, que al realizarse sobre un arreglo provocan una complejidad de **O(6)**.

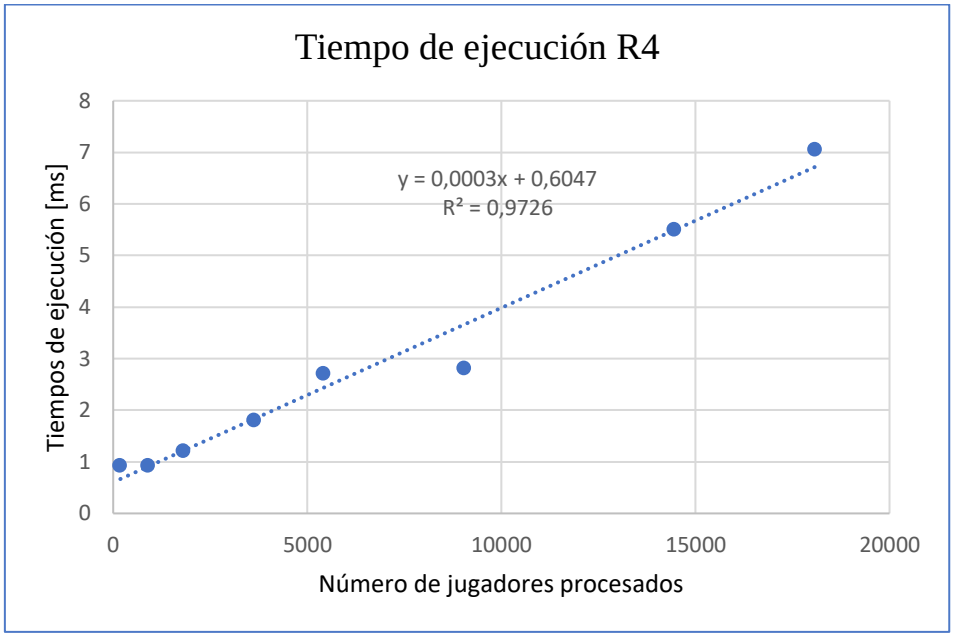
Resultando en una complejidad de **O(1 + 1 + altura + #elementos del árbol + # llaves del primer rango + 6)**.

Que se puede rescribir como **O(1 + 1 + log(# elementos del árbol) + #elementos del árbol + # llaves del primer rango + 6)**.

Y si se despeja para dejar el elemento con mayor peso se obtiene **O(#elementos del árbol)**.
Complejidad que se puede representar como **O(n)**.

Gráfica de tiempos de ejecución:

Tamaño del archivo	Número de jugadores	Tiempo de ejecución [ms]	Consumo de memoria [kB]
small	184	0.93	3.59
5pct	906	0.92	3.59
10pct	1813	1.21	3.59
20pct	3637	1.8	3.07
30pct	5427	2.71	7.71
50pct	9039	2.81	8.23
80pct	14467	5.5	5.21
large	18076	7.06	5.21



Se puede observar un buen R^2 lo cual es indicativo de que esta regresión representa de manera adecuada el fenómeno, por lo que podemos confirmar el comportamiento lineal del algoritmo implementado para el requerimiento 4. Adicionalmente, se puede evidenciar que el crecimiento de los jugadores con el comentario "Solid Player" es proporcional al tamaño del archivo que estamos utilizando.

Requerimiento 5

Entradas:

```
Numero de segmentos en que se divide el rango de propiedad en el
histograma (N)

Numero de niveles en que se dividen las marcas de jugadores en el
histograma

Propiedad de la cual se va a hacer el histograma (puede seleccionar
entre: Overall, potential, value_eur, wage_eur, age, height_cm,
weight_kg, release_clause_eur)
```

Salidas:

```
El número total de los jugadores consultados.
```

El número total de los jugadores utilizados para crear el histograma de la propiedad.

Valor mínimo y valor máximo de la propiedad consultada en el histograma.

El histograma con la distribución de los jugadores por esa propiedad.

Precarga:

```python
def R5_all_histograms(catalog, player):
    properties = ['overall', 'potential', 'value_eur', 'wage_eur', 'age', 'height_cm', 'weight_kg', 'release_clause_eur']
    for i in range(8):
        prop = properties[i]
        exist_key = om.get(catalog['R5_by_{0}'.format(prop)], player[prop])
        if exist_key is None:
            new_list = lt.newList('ARRAY_LIST')
            om.put(catalog['R5_by_{0}'.format(prop)], player[prop], new_list)
            exist_key = om.get(catalog['R5_by_{0}'.format(prop)], player[prop])

        lt.addLast(exist_key['value'], player)


def cmp_r5(p1, p2):
    x = 1
    #potencial
    if (p1 > p2):
        x= 1
    elif (p1 < p2):
        x= -1
    elif (p1 == p2):
        x = 0
    return x
```

Para la precarga y para cada uno de las propiedades que se deben poder analizar a través de los histogramas (`overall`, `potential`, `value_eur`, `wage_eur`, `age`, `height_cm`, `weight_kg`, `release_clause_eur`) se creó una Tabla de Hash donde la llave será la propiedad (‘overall’) y como valor un mapa ordenado donde se almacena el valor de la propiedad como llave y como valor el jugador (67, player1).

Análisis de complejidad:

```python
def r5_answer(catalog, in_bins, in_scale, prop):


    ordered_map = catalog['R5_by_{0}'.format(prop)] #O(1)

    #Para todos los players con un valor no vacío
    keys = om.keys(ordered_map, 0, 9999999999999999) #O(altura + num(elements))
    contador = 0
    for key in lt.iterator(keys):                          #O(numelements)
        lista_key = om.get(ordered_map, key)['value']
        contador += lt.size(lista_key)
    n_consulted = contador

    llave_max = om.maxKey(ordered_map)                     #O(altura)
    #Estas dos propiedades tienen vacíos y los vacíos están agrupados en la llave -1,
    #Por lo que se toma la siguiente llave después de -1 (el mínimo verdadero)
    if prop in ['value_eur','release_clause_eur']:
        llave_min = om.select(ordered_map, 1)
    else:
        llave_min = om.minKey(ordered_map)

    #Creación de los intervalos
    rango = llave_max - llave_min                          #O(1)
    longitud_intervalo = rango/in_bins                     #O(1)

    #Output
    ranks    = lt.newList('ARRAY_LIST')                    #O(1)
    total_elements = lt.newList('ARRAY_LIST')              #O(1)
    height = lt.newList('ARRAY_LIST')                      #O(1)
```

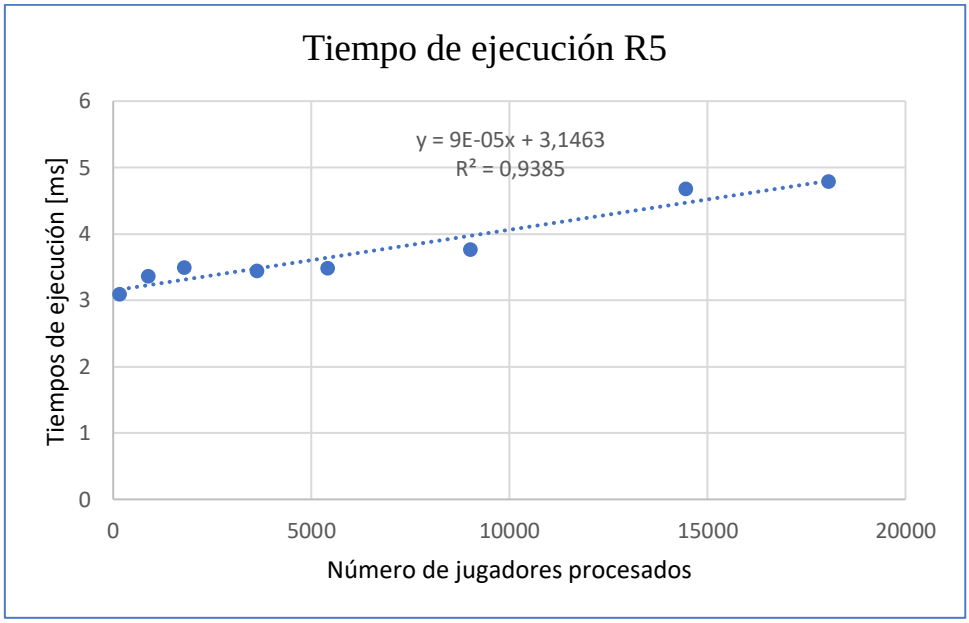
Una vez el usuario ingresa los parámetros del requerimiento, se ingresa a la Tabla de Hash correspondiente con la propiedad dada en O(1). Posteriormente, como algunas propiedades no poseen valor se decidió en la precarga ingresarles un -1 y después con om.keys() seleccionar solamente aquellos que sí tuvieran un valor, esta operación es O(altura + numelements) pero para propiedades como overall que todos los jugadores tienen valores, será más certero decir O(n). Por esto, para el análisis de este req se tomará como numelements aquellos jugadores que sí tengan valor dentro de la propiedad dada. Después, contamos cuántos jugadores no vacíos para esta propiedad hay para comparar después con la suma de intervalos esto es O(numelements). Luego, tomamos el máximo y el mínimo válido y se encuentra el rango en una complejidad de O(altura). Posteriormente, se calcula el rango y la longitud de cada intervalo, se crean 3 TAD lista auxiliares en O(1). Posteriormente, iteramos en el número de intervalos dado, calculamos la llave máxima, la llave mínima y cuántos hay dentro de este intervalo para cada intervalo. Como la suma de todos estos intervalos es igual a contar cuántos elementos hay en todo el árbol, la complejidad de toda la iteración será O(numelements). Finalmente, sumamos el número de jugadores encontrado de cada intervalo en exactamente O(bins) por lo que la complejidad final será O(numelements).

```
for i in range(in_bins):                                #O(numelements)
    intervalo_i_lo = inicio
    intervalo_i_hi = round(inicio + longitud_intervalo, 3)

    if i == (in_bins-1):
        # 93.9999... -> 94
        intervalo_i_hi = int(round(intervalo_i_hi, 1))


    if i == 0:
        #Añadir el rango
        string = '['+str(intervalo_i_lo)+', '+str(intervalo_i_hi)+']'
        lt.addLast(ranks, string)
        #llaves que aplican
        keys = om.keys(ordered_map, intervalo_i_lo, intervalo_i_hi)



    else:
        #Añadir el rango
        string = '('+str(intervalo_i_lo)+', '+str(intervalo_i_hi)+']'
        lt.addLast(ranks, string)

        #Llaves que aplican
        keys_not_fixed = om.keys(ordered_map, intervalo_i_lo, intervalo_i_hi)

        exist_key = om.get(ordered_map, intervalo_i_lo)
        if exist_key is None:
            keys = keys_not_fixed
        else:
            tam_full = lt.size(keys_not_fixed)
            keys = lt.subList(keys_not_fixed, 2, tam_full-1)
```

```
    #Contar elementos dentro del rango de llaves
    contador = 0
    for key in lt.iterator(keys):
        lista_key = om.get(ordered_map, key)['value']
        contador += lt.size(lista_key)
    lt.addLast(total_elements, contador)
    lt.addLast(height, contador//in_scale)



    #Iniciar desde i_hi la siguiente iteración
    inicio = intervalo_i_hi


n_used = 0
for i in lt.iterator(total_elements):  #O(bins)
    n_used += i



return n_consulted, n_used, llave_min, llave_max, ranks, total_elements, height
```

Gráfica de tiempos de ejecución:

Tamaño del archivo	Número de jugadores	Tiempo de ejecución [ms]	Consumo de memoria [kB]
small	184	3.09	52.16
5pct	906	3.36	54.49
10pct	1813	3.49	55.5
20pct	3637	3.44	56.23
30pct	5427	3.48	56.33
50pct	9039	3.76	56.95
80pct	14467	4.67	56.91
large	18076	4.78	56.89



Esta gráfica posee un orden de crecimiento lineal y esto se ve evidenciado por el valor de R cuadrado muy cercano a 1. Esto coincide con nuestro análisis de requerimientos, pues determinamos una complejidad de O(numelements) siendo numelements la cantidad de jugadores con valores dentro de la propiedad dada. Particularmente, los datos se hicieron con 'overall' por lo que corresponde exactamente con la cantidad de jugadores cargados, pues todos los jugadores tienen el valor de overall lleno.

Requerimiento 6

Entradas:

`Nombre corte de un jugador, Posición del jugador.`

Salidas:

`Lista con los posibles jugadores que podrían sustituir al jugador ingresado, Número de jugadores que comparten la posición dada por parámetro.`

Precarga:

```
catalog["R6_name_hash"] = mp.newMap(numelements=19000,
                                    maptype='PROBING',
                                    loadfactor=our_loadfactor,
                                    prime=our_prime,
                                    comparefunction= compareNames)

catalog["R6_min_max_for_PVAH"] = mp.newMap(numelements=19000,
                                           maptype='PROBING',
                                           loadfactor=our_loadfactor,
                                           prime=our_prime,
                                           comparefunction= comparePositions)

catalog["R6_hash_vr_scores"] = mp.newMap(numelements=21,
                                    maptype='PROBING',
                                    loadfactor=our_loadfactor,
                                    prime=our_prime,
                                    comparefunction= comparePositions)
```

Dentro de catálogo se crean 3 Tablas de Hash, todas utilizando la estructura de datos Linear Probing. En la primera, "R6_name_hash", se organizan los jugadores de tal forma que las llaves de la tabla son los nombres cortos de los jugadores y los valores son referencias a los diccionarios que contienen los distintos atributos asociados a un jugador. Esto con la finalidad de poder encontrar rápidamente al jugador que el usuario desee encontrar.

La segunda tabla de hash, "R6_min_max_for_PVAH", tiene por llaves las diferentes posiciones de juego que poseen los jugadores, y como valores una lista con 8 elementos que representan los valores máximos y mínimos para los atributos "potential", "value", "age" y "height" de todos los jugadores en una misma posición.

La última Tabla de Hash, "R6_hash_vr_scores", también tiene como llaves las posiciones de jugo, sin embargo, como valores esta vez tiene Árboles Rojo Negro con los jugadores distribuidos de acuerdo con el valor representativo que se les haya calculado.

```python
def R6_min_max_for_PVAH (catalog, player):
    positions = player["player_positions"].split(",")
    for position in positions:
        position = position.strip(" ")

        exist_pos = mp.get(catalog["R6_min_max_for_PVAH"], position)

        if exist_pos == None:
            mp.put(catalog["R6_min_max_for_PVAH"], position, lt.newList("ARRAY_LIST"))

        list_pos = mp.get(catalog["R6_min_max_for_PVAH"], position)["value"]

        if lt.isEmpty(list_pos):
            lt.addLast(list_pos, float(player["potential"])) #valor min potential 1
            lt.addLast(list_pos, float(player["potential"])) #valor max potential 2
            lt.addLast(list_pos, float(player["value_eur"])) #valor min value 3
            lt.addLast(list_pos, float(player["value_eur"])) #valor max value 4
            lt.addLast(list_pos, float(player["age"])) #valor min age 5
            lt.addLast(list_pos, float(player["age"])) #valor max age 6
            lt.addLast(list_pos, float(player["height_cm"])) #valor min height 7
            lt.addLast(list_pos, float(player["height_cm"])) #valor max height 8

        else:
            if float(player["potential"]) < lt.getElement(list_pos, 1):
                lt.changeInfo(list_pos, 1, float(player["potential"]))

            if float(player["potential"]) > lt.getElement(list_pos, 2):
                lt.changeInfo(list_pos, 2, float(player["potential"]))

            if float(player["value_eur"]) < lt.getElement(list_pos, 3):
                lt.changeInfo(list_pos, 3, float(player["value_eur"]))

            if float(player["value_eur"]) > lt.getElement(list_pos, 4):
                lt.changeInfo(list_pos, 4, float(player["value_eur"]))

            if float(player["age"]) < lt.getElement(list_pos, 5):
                lt.changeInfo(list_pos, 5, float(player["age"]))

            if float(player["age"]) > lt.getElement(list_pos, 6):
                lt.changeInfo(list_pos, 6, float(player["age"]))

            if float(player["height_cm"]) < lt.getElement(list_pos, 7):
                lt.changeInfo(list_pos, 7, float(player["height_cm"]))

            if float(player["height_cm"]) > lt.getElement(list_pos, 8):
                lt.changeInfo(list_pos, 8, float(player["height_cm"]))
```

Esta función se ejecuta al momento de agregar los jugadores, y su funcionalidad es determinar si el jugador que se tiene en cuestión posee alguno de los valores máximos para los atributos Potential, Value, Age o Height dentro de una posición especifica, en caso de hacerlo, modifica los valores, por lo que para el final de la precarga de jugadores se garantiza haber obtenido los valores máximos y mínimos de cada atributo dentro de cada posición de juego.

```python
def calc_vr(player, catalog):
    min_max_hash = catalog["R6_min_max_for_PVAH"]

    positions = player["player_positions"].split(",")
    for pos in positions:
        pos = pos.strip(" ")

        if  om.size(om.get(catalog["R2_position_trees"],pos)["value"]) > 1:

            min_max_of_pos = mp.get(min_max_hash,pos)["value"]

            min_P_pos =  lt.getElement(min_max_of_pos,1)
            max_P_pos =  lt.getElement(min_max_of_pos,2)
            min_V_pos =  lt.getElement(min_max_of_pos,3)
            max_V_pos =  lt.getElement(min_max_of_pos,4)
            min_A_pos =  lt.getElement(min_max_of_pos,5)
            max_A_pos =  lt.getElement(min_max_of_pos,6)
            min_H_pos =  lt.getElement(min_max_of_pos,7)
            max_H_pos =  lt.getElement(min_max_of_pos,8)


            vr_P = (float(player["potential"]) - min_P_pos) / (max_P_pos - min_P_pos)
            vr_V = (float(player["value_eur"]) - min_V_pos) / (max_V_pos - min_V_pos)
            vr_A = (float(player["age"]) - min_A_pos) / (max_A_pos - min_A_pos)
            vr_H = (float(player["height_cm"]) - min_H_pos) / (max_H_pos - min_H_pos)

            vr_score_in_pos = round(vr_P + vr_V + vr_A + vr_H, 5)

            exist_pos = mp.get(catalog["R6_hash_vr_scores"], pos)

            if exist_pos is None:
                new_rbt =  om.newMap(omaptype="RBT", comparefunction= compareVr)
                mp.put(catalog["R6_hash_vr_scores"], pos, new_rbt)

            ordered_map = mp.get(catalog["R6_hash_vr_scores"], pos)['value']

            exist_score = om.get(ordered_map, vr_score_in_pos)

            if exist_score is None:
                new_list = lt.newList("ARRAY_LIST")
                om.put(ordered_map, vr_score_in_pos, new_list)

            score_list = om.get(ordered_map, vr_score_in_pos)["value"]

            lt.addLast(score_list, player)
```

Una vez se han cargado todos los jugadores se ejecuta esta función, cuya finalidad es recorrer todos los jugadores, extraer las posiciones en las que juegan, calcular sus valores representativos para cada posición y finalmente agregarlos distintos Árboles Rojo Negro, cada uno representativo de una posición, donde las llaves son valores representativos y los valores son listas que contienen a todos los jugadores con un mismo valor representativo.

Análisis de complejidad:

```
def vr_for_name(player, pos, catalog):
    min_max_hash = catalog["R6_min_max_for_PVAH"]
    if  om.size(om.get(catalog["R2_position_trees"],pos)["value"]) > 1:

        min_max_of_pos = mp.get(min_max_hash,pos)["value"]

        min_P_pos =  lt.getElement(min_max_of_pos,1)
        max_P_pos =  lt.getElement(min_max_of_pos,2)
        min_V_pos =  lt.getElement(min_max_of_pos,3)
        max_V_pos =  lt.getElement(min_max_of_pos,4)
        min_A_pos =  lt.getElement(min_max_of_pos,5)
        max_A_pos =  lt.getElement(min_max_of_pos,6)
        min_H_pos =  lt.getElement(min_max_of_pos,7)
        max_H_pos =  lt.getElement(min_max_of_pos,8)


        vr_P = (float(player["potential"]) - min_P_pos) / (max_P_pos - min_P_pos)
        vr_V = (float(player["value_eur"]) - min_V_pos) / (max_V_pos - min_V_pos)
        vr_A = (float(player["age"]) - min_A_pos) / (max_A_pos - min_A_pos)
        vr_H = (float(player["height_cm"]) - min_H_pos) / (max_H_pos - min_H_pos)

        vr_score_in_pos = round(vr_P + vr_V + vr_A + vr_H, 5)

        return vr_score_in_pos
    else:
        return None
```

1) Lo primero que se hace es utilizar el nombre y la posición que ingresa el usuario para acceder al diccionario del jugador y calcular su valor representativo. Esta al ser una solución implementada bajo un Tabla de Hash poseería una complejidad de **O(1)**.

```python
def R6_answer(jugador, pos, catalog):
    tam_jugadores_in_pos = om.size(om.get(catalog["R2_position_trees"],pos)["value"])
    vr_of_player = vr_for_name(jugador, pos, catalog)
    if vr_of_player == None:
        return None, None

    exist_pos = mp.get(catalog["R6_hash_vr_scores"], pos)
    if exist_pos is None:
        return None, None

    vr_RBT_of_pos = exist_pos["value"]
    exist_list = om.get(vr_RBT_of_pos, vr_of_player)
    if exist_list is None:
        return None, None

    vr_list = exist_list["value"]
    if lt.size(vr_list) <= 1:
        var_para_volver_a_meter = om.get(vr_RBT_of_pos, vr_of_player)["value"]
        om.remove(vr_RBT_of_pos, vr_of_player)
        floor_key = om.floor(vr_RBT_of_pos, vr_of_player)
        ceiling_key = om.ceiling(vr_RBT_of_pos, vr_of_player)
        om.put(vr_RBT_of_pos,vr_of_player, var_para_volver_a_meter)
        sustitucion = None
        if floor_key == None:
            sustitucion = om.get(vr_RBT_of_pos, ceiling_key)["value"]
        elif ceiling_key == None:
            sustitucion = sustitucion = om.get(vr_RBT_of_pos, floor_key)["value"]
        else:
            if abs(floor_key - vr_of_player) > abs(ceiling_key - vr_of_player):
                sustitucion = om.get(vr_RBT_of_pos, floor_key)["value"]
            else:
                sustitucion = om.get(vr_RBT_of_pos, ceiling_key)["value"]
        return sustitucion, tam_jugadores_in_pos
    else:
        list_jugadores_validos = lt.newList("ARRAY_LIST")
        for persona in lt.iterator(vr_list):
            if persona != jugador:
                lt.addLast(list_jugadores_validos, persona)
        return list_jugadores_validos, tam_jugadores_in_pos
```

2) Una vez con el valor representativo del jugador, se verifica que exista la el árbol asociado a la posición sobre la cual se está trabajando, al hacerse uso de una Tabla de Hash, la complejidad sería **O(1)**.
3) Después de obtener el árbol se obtiene la lista asociada al valor representativo del jugador. Esta lista al estar dentro de un Árbol Rojo Negro, la complejidad de obtenerla es **O(altura).**
4) Una vez con la lista se mira cuántos elementos contiene. En caso de que tener 1 solo elemento, se sabe que ese elemento debe ser el jugador ingresado por el usuario. Por lo que se procede a borrar dicha lista del árbol, acción con complejidad **O(altura)**.
    a. Luego se hace tanto *om.ceiling()* y *om.floor()* para obtener las llaves más parecidas a la anterior. Acción con una complejidad de **O(2xaltura)**.
    b. Luego se utiliza el valor absoluto para determinar cual llave es más cercana a la original, y se retorna la lista asociada. Como se tiene que obtener la lista, esto tendría una complejidad de **O(altura)**.
5) En caso contrario, de que la lista tuviese más de 1 elemento, se recorre la lista para hallar a todos los jugadores distintos al ingresado por el usuario, se acomodan en una lista y se retornan. Esta acción tendría una complejidad de **O(# jugadores con mismo vr)**.

Por lo que se pueden tener 2 complejidades para este algoritmo:

En una se tiene **O (1 + altura + altura + 2 altura + altura)**.

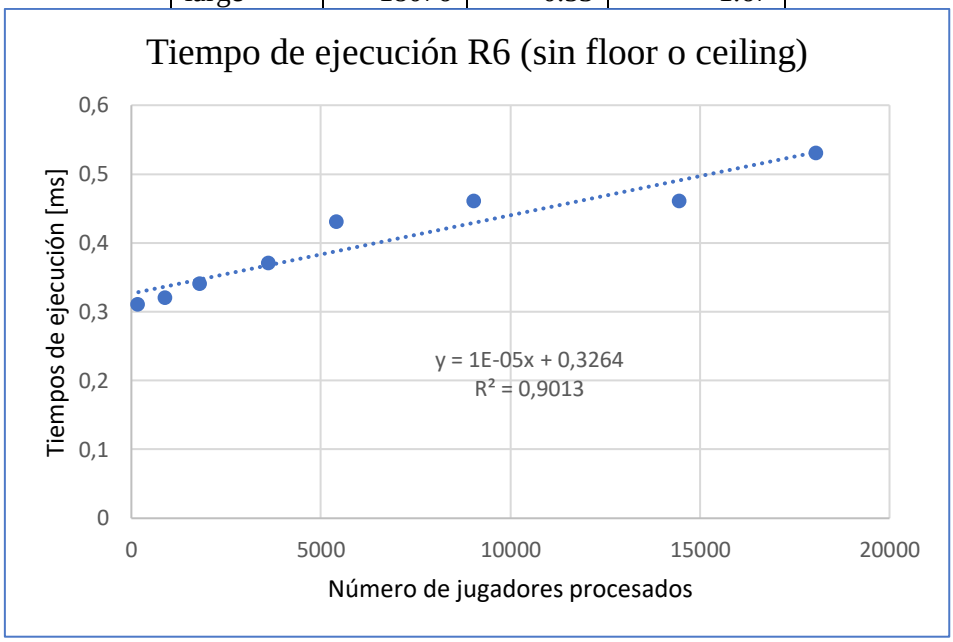
Y en la otra O**(1 + altura + # jugadores con mismo vr)**. Sin embargo, debido a que se utilizan 5 decimales para el valor representativo de un jugador, es increíblemente improbable que 2 o más jugadores compartan un mismo valor representativo, por lo que en general "# jugadores con mismos vr" será menor a la altura.

Por lo que como complejidad final queda **O(altura)** para ambos casos, la cual se puede representar de mejor manera como O(log(n)), donde n es el número de elementos que contiene el árbol.

Gráfica de tiempos de ejecución:

- Input: C. Chaplin, CAM.

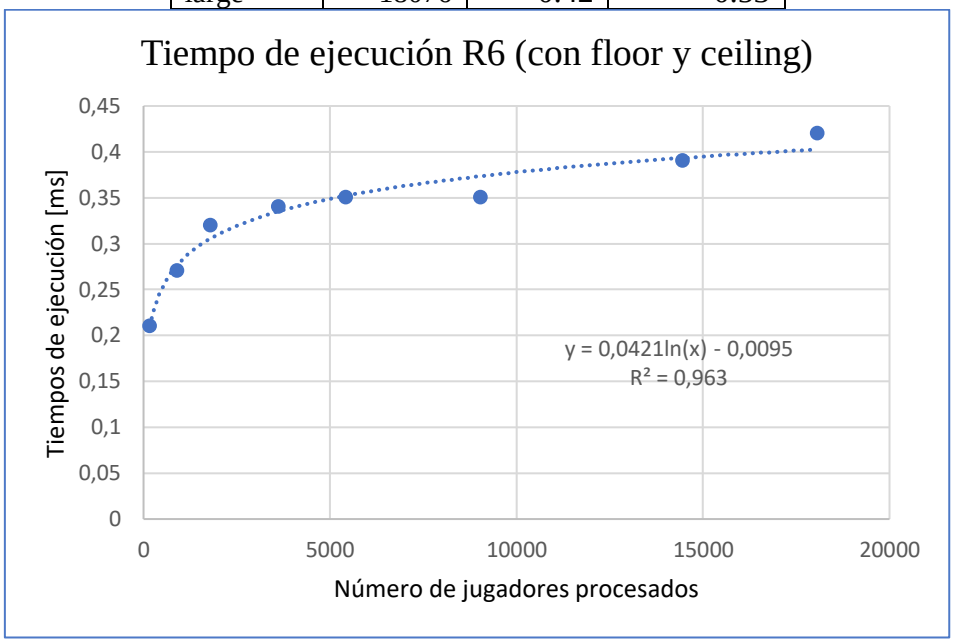
Tamaño del archivo	Número de jugadores	Tiempo de ejecución [ms]	Consumo de memoria [kB]
small	184	0.31	0.48
5pct	906	0.32	0.98
10pct	1813	0.34	2.49
20pct	3637	0.37	0.51
30pct	5427	0.43	1.67
50pct	9039	0.46	1.67
80pct	14467	0.46	7.43
large	18076	0.53	1.67



Debido a que este algoritmo puede presentar 2 comportamientos diferentes decidimos presentar ambos casos. La primera gráfica es en dónde la primera lista que se obtiene hay varios jugadores con el mismo valor representativo, por lo que para obtener las sustituciones toca recorrer la lista, el comportamiento que se exhibe tiene más similitudes al de uno línea según el R^2, sin embargo, también cabe la posibilidad de que comience a presentar un comportamiento logarítmico según se aumentasen los jugadores.

- Input: D. Zappacosta, RWB.

Tamaño del archivo	Número de jugadores	Tiempo de ejecución [ms]	Consumo de memoria [kB]
small	184	0.21	1.48
5pct	906	0.27	9.74
10pct	1813	0.32	11.2
20pct	3637	0.34	10.77
30pct	5427	0.35	10.77
50pct	9039	0.35	0.98
80pct	14467	0.39	2.01
large	18076	0.42	0.53



Por otro lado, en la segunda gráfica se utilizan los métodos *om.ceiling()* y *om.floor()* para hallar las mejores sustituciones posibles para el jugador ingresado. Como se pudo ver en el análisis de complejidad, este camino requiere de varias veces realizar operaciones con un coste de O(altura) o O(log(n)), por lo que al realizar la gráfica se puede ver como se exacerba la tendencia tener un comportamiento logarítmico.

Carga de datos

Para la carga de datos, se evidencia una complejidad lineal, con un tiempo de carga alrededor de 15s en -large.

Tamaño del archivo	Número de jugadores	Tiempo de ejecución [ms]	Consumo de memoria [kB]
small	184	205.75	2998.22
5pct	906	749.81	12237.02
10pct	1813	1561.12	23610.47

20pct	3637	2849.8	46322.8
30pct	5427	4280.84	68413.73
50pct	9039	7346	112877.83
80pct	14467	10889.25	179235.62
large	18076	13264	223236.08