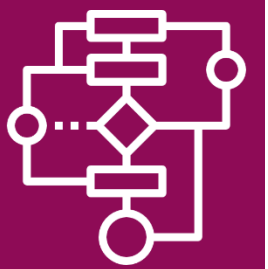
UNIVERSITAT DE LLEIDA

ESCOLA POLITÈCNICA SUPERIOR

ALGORÍTMICA Y COMPLEJIDAD

ASIGNADOR DE MESAS

PRIMERA PRÁCTICA



Docent: Aitor Corchero Rodriguez
Alumnos: Mohamed Largo Yagoubi
Theo Moreno Lomero
DNIs: 46164842R – 45722473Z

Grados: GEI - GEIADE
Grupos: A - B
Curso: 2021-2022

1 ANÁLISIS DE COSTES TEORÍCOS

1.1 ANÁLISIS DE LA FUNCIÓN ITERATIVA

```
def bookerineManagement_iterativo(reserves): # O(n·log(n) + n)
  k = 1 # O(1)
  reserves.sort(key=lambda x: int(x[1])) # O(n·log(n))
  if int(reserves[0][1]) != 0: # O(1)
    return 0 # O(1)
  for j in range(len(reserves)): # O(n)
    if k == len(reserves) or int(reserves[k][1]) - int(reserves[j][1]) > 1:#
O(1)
      return int(reserves[j][1]) + 1 # O(1)
    j += 1 # O(1)
    k += 1 # O(1)
  return int(reserves[-1][1]) + 1 # O(1)
```

Por lo tanto, el coste de nuestra función iterativa será de **O(n·log(n)+n)** ya que la función **sort()** en el peor de los casos realizará **O(n·log(n))** iteraciones para ordenar a lista, además concatenamos un bucle while que recorre toda la lista hasta encontrar la primera mesa libre, en el peor de los casos tendrá que recorrer toda la lista de n elementos, es decir **O(n)**.

1.2 ANÁLISIS DE LA FUNCIÓN RECURSIVA

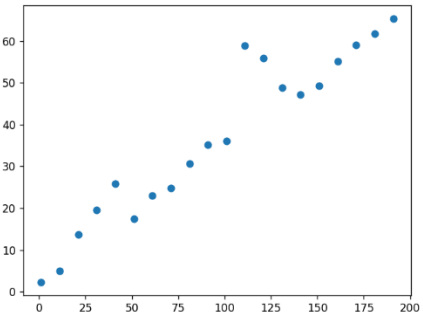
```
def bookerineManagement_recursivo(reserves): # O(n·log(n) + n)
 reserves.sort(key=lambda x: int(x[1])) # O(n·log(n))
    return bookerineManagement_recursivo_cola(reserves, 0) # O(1)

def bookerineManagement_recursivo_cola(reserves, res): # O(n)
    if not reserves: # O(1)
        return res # O(1)
    if int(reserves[0][1]) > res: # O(1)
        return res # O(1)
    return bookerineManagement_recursivo_cola(reserves[1:],int(reserves[0][1])
+ 1) # O(n)
```

En la función recursiva también se ha utilizado la función **sort()** ya implementada de python, que como se ha explicado en el apartado anterior se obtendrá un coste de **O(n·log(n))** en el peor de los casos. Para realizar la función recursiva se ha implementado una función auxiliar de tal forma hacer la recursividad del problema, para obtener el caso simple de la función esta tendrá que llamar n veces la longitud de la lista, es decir que tendrá que recorrer toda la lista hasta encontrar la primera mesa libre, por lo tanto, el coste de la función auxiliar es de **O(n)**, el cómputo total de la complexidad será de **O(n·log(n)+n)**.

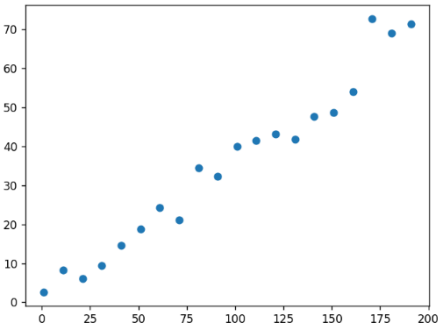
2 ANÁLISIS DE COSTES EMPÍRICOS

2.1 ANÁLISIS DE LAS FUNCIONES ITERATIVA Y RECURSIVA

Número de reservas 10

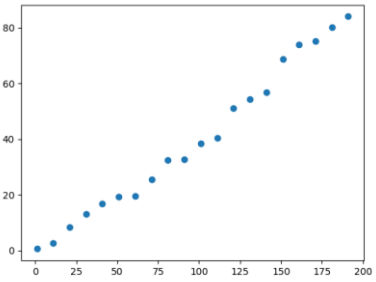


Iterativo

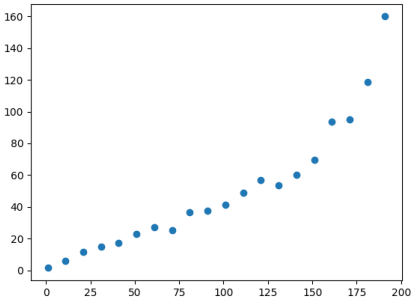


Recursivo

Número de reservas 100

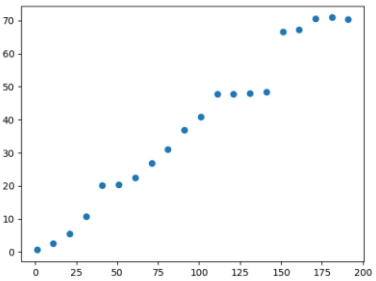


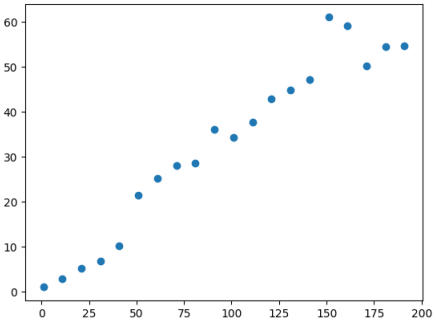
Iterativo



Recursivo

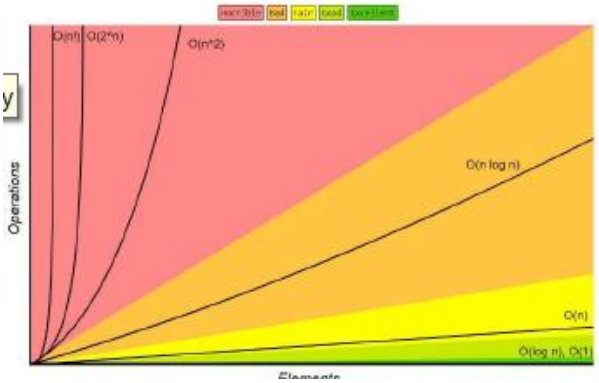
Número de reservas 1000





Iterativo *Recursivo*

Utilizando esta grafica de referencia podemos observar que, para las reservas de 100, 1000 en iterativo se obtiene una curva de O(n · log n), en cambio para 10 reservas podemos ver su curva en algunos de sus puntos tiende a obtener un coste de $O(n^2)$.



Si nos fijamos en el caso de las gráficas de la función recursiva, podemos ver que también suceden cosas similares, para 100 reservas vemos que su curva obtiene un coste de O(n·log n), y para 10,100 reservas podemos obtener una curva en donde diferentes puntos tiende a un coste $O(n^2)$.

Por lo tanto podemos deducir que tanto la función iterativa como recursiva tienen un coste algorítmico que se encuentra acotado superiormente por $O(n^2)$ e inferiormente por O(n · log n), es decir es cercano a O(n · log n + n).

3 DESCRIPCIÓN DEL PSEUDOCÓDIGO

3.1 PSEUDOCÓDIGO DE LA FUNCIÓN ITERATIVA

```
1  Procedimiento ObtenerNúmeroDeMesaIterativo(reservas)
2  | // Comentario: reservas = [('Name1','Num1'),('Name2','Num2'), ... ,('NameN','NumN')] es una
   |    lista de n tuplas
3  | k ← 0;
4  | reservas.ordenar(de menor a mayor número de mesa, Integer <-- (String)tupla[1])
5  | // Comentario: Ordenamiento de la lista de menor a mayor número de mesa mediante la función
   |    sort y una función lambda. Realizar casteo de String a Entero del segundo valor de cada tupla
   |    de la lista (el número de mesa)
6  | SI Integer ← (String)reservas[0][1] != 0 Entonces
7  | | // Comentario: Si el número de mesa del primer elemento de la lista ordenada no es un cero
8  | | retornar 0
9  | FIN SI
10 | Para Cada j ← 0 < |reservas| hasta n Con Paso 1 Hacer
11 | | // Comentario: Bucle que recorra elemento por elemento la lista de reservas
12 | | SI k == tamaño(reservas) V Integer ← (String)reservas[k][1] - Integer ← (String)reservas[j][1] > 1
13 | | | // Comentario: Si se ha llegado al último elemento de la lista o dado un elemento este sea más
   | | |    grande en dos unidades o más que su antecesor
14 | | | retornar Integer ← (String)reservas[j][1] + 1
15 | | | // Retornar el antecesor más una unidad
16 | |  FIN SI
17 | | j ← j + 1
18 | |  k ← k + 1
19 | FIN Para Cada
```

```
20 | retornar Integer ← (String)reservas[-1][1] + 1
21 | // Retornar el último número de mesa más una unidad
22 FIN Procedimiento
```

3.2 PSEUDOCÓDIGO DE LA FUNCIÓN RECURSIVA

```
1  Procedimiento ObtenerNúmeroDeMesaRecursivo(reservas)
2  | //Comentario: reservas = [('Name1','Num1'),('Name2','Num2'), ... ,('NameN','NumN')] es una
3  | lista de n tuplas
4  | reservas.ordenar(de menor a mayor número de mesa, Integer <-- (String)tupla[1])
5  | // Comentario: Ordenamiento de la lista de menor a mayor número de mesa mediante la función
   | sort y una función lambda. Realizar casteo de String a Entero del segundo valor de cada tupla
   | de la lista (el número de mesa)
6  | retornar Integer ← bookerineManagement_recursivo_cola(reservas,0)
7  | // Comentario: devuelme el resultado de la llamada recursiva
8  | Procedimiento bookerineManagement_recursivo_cola(reservas,res)
9  | | SI not reserves Entonces
10 | | | // Comentario: Para llegar a este caso simple hay dos maneras, la primera manera es cuando
   | | |    no tengamos ninguna reserva, y la segunda es cuando hallamos recorrido toda la lista.
11 | | | retorna res
12 | | FIN SI
13 | | SI Integer ← (String)reservas[0][1] > res  Entonces
14 | | | // Comentario: Se comprueba si el primer elemento de la lista es mayor al resultado, este
   | | | resultado se trata del primer elemento de la lista +1, te tal forma que cuando tengamos una
   | | | lista con n que no sean secuenciales devolverá el resultado anterior
13 | | | retorna res
14 | | FIN SI
15 | | retorna Integer ← bookerineManagement_recursivo_cola((String )reserves[1:],
   | | (String)reserves[0][1]) + 1)
16 | | // Comentario: devolvemos la lista reservas a excepción del primer elemento i rescribimos la
   | | variable res
17 | FIN Procedimiento
18 FIN Procedimiento
```

4 CONVERSIÓN DEL ITERATIVO A RECURSIVO

Para poder pasar de iterativo a recursivo lo primero que hacemos es fijarnos en el caso simple que utiliza la función iterativa, en ambas funciones los que hacemos es comprobar si la primera posición de la lista es decir la primera mesa reservada se trata de la mesa 0, por lo tanto, si es diferente al número 0 de tal forma podemos saber si esa mesa está reservada.

En la segunda parte de la función iterativa utilizamos un bucle for de tal manera nos permite recorrer toda la lista, como sabemos que ese bucle se ejecutará n veces la longitud de la lista podemos deducir que la función recursiva deberá llamarse n veces la longitud de la lista hasta llegar al caso simple, este problema solo sucederá en ambas en el peor de los casos.

5 CONCLUSIÓN

Aunque no hayamos logrado obtener un coste lineal, creemos que nuestra implementación es la más optima ya que aprovecha mejor los espacios de

memoria, en un primer momento teníamos un algoritmo más eficiente de coste O(n) que consistiría en crear una lista auxiliar de longitud igual al elemento máximo más uno e ir colocando cada elemento en su respectiva posición y mediante un bucle que empiece desde la posición 0 recorra elemento por elemento hasta encontrar la primera posición vacía o número de mesa libre, ejemplo: si tuviésemos la lista siguiente [(“Laura”,”200”), (“Carlos-Reserva”,”20”)] el principal problema es obvio ya que tendríamos que crear una lista de 200 + 1 posiciones y solo estaríamos utilizando dos de ellas, por ello creemos que esta solución no es la más eficiente pero si la más óptima. Por otra parte, se nos ocurrió otro algoritmo que evitaba el uso de una lista auxiliar y el hecho de tener que ordenar la lista actual, consistía en buscar mediante un bucle el elemento mínimo 0 después el 1 y así repetidamente hasta encontrar el primer elemento consecutivo que no apareciese en la lista, el problema es evidente en el peor de los casos sería un coste de O(n!), peor que el coste que tenemos, por lo tanto, la solución que proponemos es óptima pero no es ni la más ni la menos eficiente.