FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 1

[75.29 / 95.06] Teoría de Algoritmos I
Segundo cuatrimestre de 2023
Primera Entrega

Alumno	Padrón	Email
Martin Pata Fraile de Manterola	106226	mpata@fi.uba.ar
Franco Daniel Capra	99642	fcapra@fi.uba.ar
Mateo Bulnes	106211	mbulnes@fi.uba.ar
Carolina Pico	105098	cpico@fi.uba.ar

Índice

1. Parte 1:

1.1. Enunciado

Nos informan de la apertura de una nueva base de investigación antártica. En la misma se espera realizar una serie de experimentos. Por lo tanto, han comenzado una búsqueda de personal calificado. Se cuenta con un listado de "n" habilidades a cubrir por el personal (Ejemplo: "Cocinar", "Primeros auxilios", "Meteorología", "astronomía", "Electricidad", etc). Además se ha reunido una cantidad de "m" candidatos. Cada candidato cubre un subconjunto de las habilidades. Nos solicitan que los ayudemos a resolver el problema intentando seleccionar a la menor cantidad de expedicionarios, sin dejar de cubrir los requerimientos.

1.2. Solución con Branch and Bound

Se cuenta con una lista de habilidades a cubrir y una lista que contiene candidatos donde, para cada uno de ellos, hay una lista con las habilidades que cubre (Para las habilidades solo se usaron sus identificadores numéricos).
A cada candidato se lo ordena por la cantidad de habilidades que cubre de manera descendente. Que un candidato cubra una mayor cantidad de habilidades no significa que genere mayor ganancia, dado que esta depende de si parte de las habilidades que cubre actualmente ya han sido cubiertas.

Utilizaremos una estructura de árbol binario, donde cada nivel corresponde a determinar si el candidato actual se incluye o no dentro del conjunto de candidatos solución. La raíz corresponde a no tener ningún candidato seleccionado y todas las habilidades por cubrir. Cada nodo en el nivel ï"tiene dos descendientes. Si tenemos n candidatos, la profundidad máxima corresponderá a n. La función costo "l"será definida por las habilidades nuevas que aporta el candidato, si el candidato aporta al menos una habilidad nueva, lo incluye. Luego l = `habilidades_candidato - hab_cubiertas`.
Para recorrer el árbol comenzamos por la raíz. Dado el nodo actual, expandimos sus descendientes y calculamos su función costo. Dada la rama del árbol que incluye o no al candidato, se compara cuál de las dos genera mejor ganancia, es decir la que nos brinda la menor cantidad de candidatos ya que queremos la mínima combinación de candidatos que cubran la lista de habilidades.
Seleccionamos al mejor de ellos y definimos al nodo actual. Si no quedan descendientes por explorar se regresa al nodo padre. A su vez, si ya se recorrieron todos los candidatos, se indica que no hay una combinación de candidatos que pueda cubrir la lista de habilidades.

De esta forma, se contemplarán tanto el caso en el que el candidato actual se incluya como el caso de no incluirlo, para posteriormente comparar ambas soluciones y elegir la que menos candidatos tiene. En caso de que las habilidades del candidato actual ya estén cubiertas, no se lo incluye y evitamos recorrer esa rama del árbol.

1.3. Pseudocódigo

```
sea habilidades la lista que contiene las habilidades
sea candidatos la lista de candidatos, donde cada candidato tiene
   una lista de habilidades
sea nro cand actual la posici n en el listado de candidatos a
   evaluar
sea hab cubiertas es la lista que contiene las habilidades que se
   van cubriendo
sea incluidos la lista de candidatos incluyendo al candidato actual
sea excluidos la lista de candidatos excluyendo al candidato actual
```

```
branch and bound(nro cand actual, hab cubiertas)
  si hab cubiertas == habilidades
    devolver candidatos seleccionados
  si nro cand actual == n
    devolver no encontre solucion

sea habilidades candidato las habilidades que aporta el nuevo
   candidato

si habilidades candidato - hab cubiertas > 0
  incluidos = branch and bound(nro cand actual +1, hab cubiertas U
   habilidades candidato)
  si incluidos no es vaci
    agregar candidato a incluidos
excluidos = branch and bound(nro cand actual +1, hab cubiertas)


si excluidos es vaci  o incluidos no es vac o y tiene menos
   candidatos que excluidos
  devolver incluidos
si no
  devolver excluidos
```

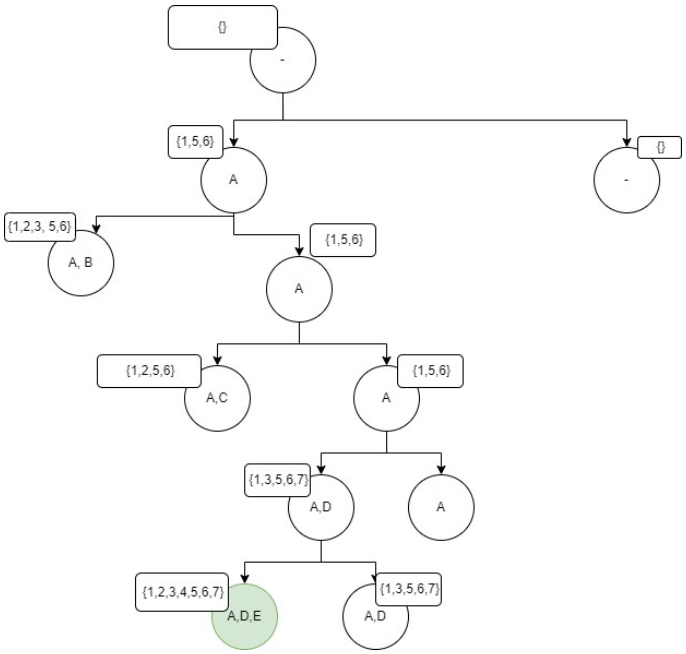
1.4. Complejidad

Con respecto a la complejidad temporal, en el peor de los casos se analizan 2^n estados del problema, se realizan cálculos de O(1) y actualizar la mejor solución encontrada es O(n), por lo que la complejidad temporal corresponde a $O(2^n * n)$, donde n corresponde a la cantidad de candidatos evaluados

Con respecto a la complejidad espacial, la memoria utilizada es proporcional a la profundidad máxima posible que es "n", por lo que dicha complejidad corresponde a O(n)

1.5. Ejemplo Paso a Paso de la solución

Contamos con una lista de 7 habilidades y debemos seleccionar de los siguientes candidatos la mínima cantidad que cubra todas las habilidades: [(A, 1, 5, 6), (C, 1, 2), (D, 3, 7), (E, 2, 4), (F, 1, 3), (B, 2, 3, 6),]. Ordenamos la lista de la forma: [(A, 1, 5, 6), (B, 2, 3, 6), (C, 1, 2), (D, 3, 7), (E, 2, 4), (F, 1, 3)].



1.6. Complejidad Solución Programada

Analizando se ve que la función `branch_and_bound` se llama recursivamente para cada candidato y en cada llamada, se realizan dos llamadas adicionales a la función con diferentes combinaciones de candidatos incluidos y excluidos. En el peor de los casos, la función debe explorar todas las combinaciones posibles de candidatos, por lo que obtenemos una complejidad temporal $O(2^n)$, donde **n** corresponde a la cantidad de candidatos.
Por otro lado, ambas condiciones de corte (si ya se cubrieron todas las habilidades o si ya se recorrieron todos los candidatos) se evalúan 1 sola vez por lo que tienen complejidad O(1).
Finalmente la decisión de cual de los 2 conjuntos tomar (incluidos o excluidos), tiene complejidad O(1) ya que es solo evaluar y retornar. Por lo tanto podemos ver que la complejidad final es $O(2^n)$, a diferencia de lo hallado desde el punto de vista teórico $O(2^n * n)$.

En cuanto a la complejidad espacial, la memoria a utilizar corresponderá con la profundidad máxima que se puede alcanzar, por lo que en el peor de los casos será n, que corresponde a la cantidad de candidatos. Es por esto que, al igual que en el análisis teórico, la complejidad espacial será O(n)

2. Parte 2:

2.1. Enunciado

Nos proponen el siguiente juego de cartas en el que tenemos que adivinar la carta que tiene un rival. El mazo tiene 1 carta de “1 de Oro”, 2 cartas de “2 de Oro” y así hasta 9 cartas de “9 de Oro”.

El rival mezcla y selecciona una carta. Mediante preguntas que solo se pueden responder por sí o por no tenemos que averiguar en la menor cantidad de consultas cual es la carta. (ejemplos: “La carta es mayor a 4?, “La carta es un ‘1’ o un ‘3’?”, etc).

Proponer un algoritmo que resuelva el problema minimizando la cantidad probable de preguntas a realizar.

2.2. Posibles soluciones Greedy

Para el problema planteado, se podrian tomar diferentes elecciones codiciosas para aplicar la metodologia Greedy.

La primera podria ser utilizando como eleccion la carta que mas probabilidad tiene de ser elegida y repetir este procedimiento en cada nueva iteracion. Esto daria como resultado la siguiente cantidad de preguntas para cada carta elegida:

Carta	Preguntas
9	1
8	2
7	3
6	4
5	5
4	6
3	7
2	8
1	8

Lo cual daria una esperanza de 3.64 (4) preguntas para adivinar la carta.

Otra alternativa seria elegir las dos cartas con mas probabilidades de ser elegidas con lo cual obtendríamos:

Carta	Preguntas
9	2
8	2
7	3
6	3
5	4
4	4
3	5
2	5
1	4

Lo cual daría una esperanza de 3.066 (4) preguntas para adivinar la carta.

Esto representa una mejora en la cantidad de preguntas pero no se puede generalizar y tampoco seria la solución óptima.

Si se analiza el problema se puede notar dos características fundamentales para resolver de forma óptima este problema mediante la metodología Greedy.

- La primera consiste en que siempre se debería preguntar lo antes posible por la carta con una mayor probabilidad, pero si se pregunta solo por un conjunto de cartas antes, esto suma una pregunta a las siguientes. Por ejemplo, si se pregunta únicamente por el 9, cualquier otra carta va a necesitar dos preguntas (la pregunta del 9 y su respectiva pregunta).
- El otro compartimiento que se puede notar es que si se agrupa más de una carta o no se agrupan las cartas, la siguiente carta con mas probabilidades no va a requerir una pregunta extra. Por ejemplo, si se toman las cartas del 9 al 6, si se pregunta por el 9 únicamente, el 8 requerirá, según cómo se agrupe, al menos dos preguntas. Si se pregunta por el 9 y el 8, el 8 volverá a necesitar por lo menos 2 preguntas.

Con esto se puede notar como a la siguiente carta no le influye la agrupación. Por lo tanto, se debería elegir la carta o grupo de cartas que tengan mas probabilidades de ser la carta elegida sin contar la siguiente carta mas probable.

En el ejemplo de tomar cartas del 9 al 6, como el 9 no tiene mas probabilidades que el grupo del 7 y el 6, tenemos que agrupar al 9 con el 8. De esa forma todas las cartas se pueden determinar en 2 preguntas. cualquier otra forma de establecer las preguntas requerirían 2 o mas de 2.

A partir de esto, se establecerá el criterio Greedy que se va a utilizar.

La elección Greeedy va a basarse en si el conjunto de cartas actual es mayor al siguiente conjunto sin contar la próxima carta con mayor probabilidad.
Si esto se cumple se hace la pregunta por ese conjunto, sino se suma la próxima carta con mayor probabilidad y se repite el procedimiento.

2.3. Ejemplo paso a paso

Aplicando el criterio previamente mencionado, se deberían realizar los siguientes pasos.

1. Obtener todas las cartas con sus respectiva cantidad de repeticiones.
2. Ordenar las cartas de mayor a menor según las repeticiones.
3. Tomar la carta con mayor cantidad de repeticiones y agregarlo al conjunto 1.
4. Sumar las repeticiones del resto de cartas sin contar la inmediatamente siguiente.
5. Comparar si la cantidad de repeticiones del conjunto 1 con las del resto de cartas sumadas.

   Si es mayor, hacer la pregunta por el conjunto 1. Sino, repetir desde el punto 3.
6. Si la respuesta a la pregunta es ‘si‘, repetir desde el punto 3 con el conjunto 1. Sino, repetir con el resto de cartas descartando el conjunto 1.

Este procedimiento genera:

Carta	Preguntas
9	2
8	3
7	3
6	3
5	3
4	3
3	4
2	5
1	5

Esto genera una esperanza de 3 preguntas, la menor de todas las que se analizaron.

2.3.1. Pseudocodigo

```
Sea valor un array con los valores de las cartas.

Sea cantidad un array con la cantidad de cartas 1 a 1 con valor.

total = suma de cantidad
```

```
Ordenar cantidad de forma descendente.

Ordenar valor de forma descendente segun los valores de cantidad.

Mientras el largo de valor <> 1:
    izquierda = 0
    derecha = total - cantidad [0] - cantidad[1]

    i = 1
    Mientras izquierda < derecha && i < largo cantidad - 2:
        izquierda += cantidad[i]
        derecha -= cantidad[i + 1]
        i++

    derecha += cantidad[i]

    si la carta esta en valores[i:]:
        cantidad = cantidad[i:]
        valores = valores[i:]
        total = derecha
    sino:
        cantidad = cantidad[:i]
        valores = valores[:i]
        total = izquierda

Mostrar el unico valor de valor
```

2.3.2. Complejidad

El pseudocódigo anterior tiene una complejidad temporal de O(1) y de complejidad espacial O(1) debido a que solo se guardan valores de 1 a 9 y solo se ordenan esos valores sin importar la cantidad de cartas de cada numero que se use.

Si en lugar de elegirse cartas del 1 al 9 fuesen de 1 a n, la complejidad temporal seria O(n log n) correspondiente al ordenamiento y de complejidad espacial O(n).

2.4. Solución Greedy

Al intentarse resolver este problema mediante la metodologia Greedy tiene que cumplirse dos condiciones:

- El problema debe contar con una subestructura óptima. Es decir, debe llegarse a la solución del problema mediante la elección codiciosa para la resolución de cada subproblema.
- Debe existir un criterio para realizar una elección codiciosa en cada subproblema.

2.4.1. Elección codiciosa

La estrategia de selección codiciosa que utilizaremos consiste en elegir una pregunta que se refiera a si una carta pertenece a un conjunto de cartas específico. Este conjunto de cartas especifico esta compuesto por aquellas que cumplan las siguientes condiciones:

- Las cartas deben estar ordenadas de forma descendente según el numero de repeticiones.

- Se eligen las cartas contiguas de mayor cantidad de repeticiones (con menor cantidad de elementos), que generen un grupo que, sumando todas las repeticiones, sea mayor al resto de la suma de repeticiones de las otras cartas, sin contar la carta con mayor cantidad de repeticiones de este ultimo grupo.

2.4.2. Subestructura óptima

La demostración de la optimalidad de la solución se va a realizar mediante reducción al absurdo.

Si se llama ‘SP‘a la solución planteada y ‘SO‘ a la solución óptima, se demostrara que ‘SP‘ es igual o mejor a ‘SO‘.

Si se generaliza el conjunto de cartas del enunciado, se tiene un conjunto de ‘1‘ a ‘N‘ cartas con un ‘1‘, dos ‘2‘, etc.

‘SO‘ se va a diferenciar de ‘SP‘ al tener que realizar al menos una pregunta menos para una carta ‘M‘. Por lo tanto, ‘M‘ tiene que ser el ultimo elemento de un subconjunto ‘N‘ a ‘M‘ o el primero de un subconjunto ‘M‘ a ‘1‘.

Como ‘M‘ es el primer elemento que difiere en cantidad de preguntas, no puede ser el ultimo elemento de un subconjunto ya que el ultimo elemento de un subconjunto de ‘SP‘ tiene la misma cantidad de preguntas que el anteúltimo elemento y por lo tanto, ‘M‘ no podría ser el primer elemento en diferir a menos que su anterior haya diferido, lo que lleva a un absurdo.

Si en cambio ‘M‘ es el primer elemento de un subconjunto, ‘SP‘ lo seleccionaría como primer pregunta si la cantidad de repeticiones de ‘M‘ es mayor a la del resto de cartas, sin contar la inmediatamente siguiente.

Entonces, si ‘M‘ cumple con ese criterio, ‘SP‘ haría la misma cantidad de preguntas por ‘M‘ que ‘SO‘ o ‘SO‘ haría mas cantidad de preguntas por el subconjunto en general con lo cual ‘SO‘ no optimizaría este subconjunto lo que seria un absurdo.

En caso contrario, ‘M‘ volvería a ser el primer elemento de un subconjunto repitiendo el caso anterior.

3. Parte 3:

3.1. Enunciado

Se está por realizar un concurso de conocimientos en parejas para escuelas secundarias. Existen "n" categorías que se evaluarán en el mismo. Una escuela evaluó a sus posibles participantes. Por cada uno de ellos generaron una lista ordenada de mayor a menor de las categorías según sus conocimientos. En base a una competencia interna se seleccionó a uno de ellos como el capitán. Nos solicitan que los ayudemos, basándonos en el concepto de inversión, a seleccionar a otro participante que mejor se complemente con el capitán.

Se pide:

1. Proponer un algoritmo utilizando división y conquista que lo resuelva. Incluir pseudocódigo
2. Analice la complejidad del algoritmo utilizando el teorema maestro y desenrollando la recurrencia.
3. Brindar un ejemplo de funcionamiento.
4. Programe su solución
5. Analice si la complejidad de su programa es equivalente a la expuesta en el punto 2.

3.2. Pseudocodigo

Para solucionar el problema de inversión mediante división y conquista vamos a tener que utilizar merge sort, pero ya que se requiere encontrar la mejor pareja para el capitán vamos a tener que poner otro orden numérico que no es de menor a mayor. La mejor pareja para el capitán va a ser ella que tenga mayor conocimiento en la categoría que mas sabe el capitán y así con el resto de las categorías. Entonces para imponer ese orden vamos a usar un diccionario que tenga como llave el conocimiento y como valor el índice donde tiene que ir. Por ejemplo si el capitán tiene los siguientes conocimientos de mayor a menor `[3,1,2,4]` el diccionario va a ser {`4:0,2:1,1:2,3:3`}.

```
// P es una Lista con las habilidades de los participantes
// C es una Lista de las habilidades del capitan
Buscar-Mejor-Compa(P:Lista[Lista],C:Lista):
    O = Crear-Orden(C)
    min_inv = INFINITO
    resultado = -1
    PARA cada (i,X) en P:
        (inv,_) = Ordenar-Contar-Con-Orden(X,O)
        SI inv < min_inv:
            min_inv = inv
            resultado = i
    RETORNAR i

Crear-Orden(C:Lista):
    Sea O diccionario
    Sea L la C inverrtido
    i = 0
    Mientras i < |L|
        O[L[i]] = i
    RETORNAR O
```

```
Ordenar-Contar(L:Lista, Orden:Diccionario):
    Si |L|=1
        RETORNAR (0,L)

    Sea A la primera mitad de L
    Sea B la segundo mitad de L
    (ia,A) = Ordenar-Contar(A, Orden)
    (ib,B) = Ordenar-Contar(B, Orden)
    (i,L) = Merge-Contar(A, B, Orden)

    RETORNAR (i+ia+ib,L)

Merge-Contar(A:Lista, B:Lista, Orden:Diccionario):
    Sea L lista
    inv = j = i = 0
    MIENTRAS i<|A| y j<|B|:
        a = Orden[A[i]]
        b = Orden[B[j]]
        Si a>b
            L[i+j]=a , i++
        Sino
            L[i+j]=b , j++
            inv += (|A|- i)

    MIENTRAS i < |A|
        L[i+j]=A[i]
    MIENTRAS j < |B|
        L[i+j]=B[i]

    RETORNAR (inv,L)
```

Este pseudocódigo esta inspirado en lo visto en la clase de Contando Inversiones

3.3. Análisis de complejidad

Para analizar la complejidad de este algoritmo vamos a tener 2 partes. La primera vamos a calcular la complejidad de Ordenar-Contar-Con-Orden con el teorema maestro, la segunda parte vamos a hacer el análisis completo del algoritmo que encuentra la mejor pareja para el capitán.

3.3.1. Calculo del complejidad de Ordenar-Contar-Orden

Para calcular la complejidad del Ordenar-Contar-Orden primero vamos a tener que calcular la recurrencia:

$$T(n) = aT(n/b) + O(n^c) \tag{1}$$

Donde:

- a: Cantidad de llamadas Recursivas
- b: proporción del tamaño original con el que llamamos recursivamente
- c: el costo de partir y juntar (Todo lo que no es llamadas recursivas)

En Ordenar-Contar entera una lista de n elementos, en este se hacen 2 llamados recursivos de con la mitad de elementos por lo tanto $a = b = 2$. Solo queda calcular la complejidad del llamado no recursivo a Merge-Contar, en ese se recorren n elementos ya que es $A + B = n/2 + n/2$ y se hace un llamado a un diccionario que es $O(1)$, entonces queda $o(n)$

Con todo lo analizado anteriormente la ecuación de recurrencia es:

$$T(n) = 2T(n/2) + O(n) \tag{2}$$

Para calcular la complejidad vamos a utilizar el Teorema Maestro que dice si tenemos un algoritmo cuya recurrencia es $T(n) = AT(n/B) + O(n^C)$

- Si $\log_B(A) < C \Rightarrow T(n) = O(n^C)$
- Si $\log_B(A) = C \Rightarrow T(n) = O(n^C \log(n))$
- Si $\log_B(A) > C \Rightarrow T(n) = O(n^{\log_B(n)})$

Para Ordenar-Contar $\log_2 2 = 1 = C$ entonces su complejidad es $O(n \log(n))$

3.3.2. Calculo Total de la Complejidad

En Buscar-Mejor-Compa tenemos que ingresan m participantes con n categorías (Matriz $m*n$) y las n categorías del capitán.

Primero se crea el orden donde se invierte una lista $O(n)$ y después recorriendo esa lista se las mete en un diccionario. Meterlas al diccionario siempre va a ser $O(1)$ ya que todas la categorías tienen que ser distintas o no tendría sentido, por lo tanto la creación del orden es $O(n)+O(n) = O(n)$.

Después tenemos el calculo de las inversiones donde se recorren los m participantes y se llama a Ordenar-Contar-Con-Orden con sus n habilidades. Con esto nos queda que la complejidad seria $O(m * n \log(n)) = O(n^2 \log(n))$ y ya que eso es mayor a $O(n)$

$$\Rightarrow \boxed{T(n) = O(n^2 \log(n))} \tag{3}$$

3.4. Ejemplo del funcionamiento

Para este ejemplo vamos a tener 3 participantes y 5 categorías.

```
Capitan = [4,5,2,1,3]

Participantes = [
    [5, 2, 1, 4, 3],  //Participante 0
    [1, 3, 2, 4, 5],  //Participante 1
    [2, 5, 3, 1, 4],  //Participante 2
]
```

Lo primero que se hace es crear el orden, para ello vamos invertir la lista del capitan y meter la categorías con los indices esperados

```
Capitan = [4,5,2,1,3]

Crear-Orden([4, 5, 2, 1, 3]):
    Sea O diccionario
    L = [3, 1, 2, 5, 4]
    i = 0
    Mientras i < |L|
        O[L[i]] = i
    RETORNAR O = {3:0, 1:1, 2:2, 5:3, 4:4}
```

Después se va a llamar para cada participante Ordenar-Contar para contar las inversiones hasta llegar al mejor orden. Demostrémoslo con el primer participante $[5, 4, 1, 2, 3]$ y el Orden **O**

3.4.1. Análisis primer participante

Primero hagamos el análisis de las llamadas recursivas donde se divide el problema

1. `Ordenar-Contar([5,2,1,4,3],O)` se separa en 2 llamados recursivos:
   `Ordenar-Contar([5,2],O)` se separa en 2 llamados recursivos:
   `Ordenar-Contar([5],O)`
   `Ordenar-Contar([2],O)`
   `Ordenar-Contar([1,4,3],O)` se separa en 2 llamados recursivos:
   `Ordenar-Contar([1],O)`
   `Ordenar-Contar([4,3],O)` se separa en 2 llamados recursivos:
   `Ordenar-Contar([4],O)`
   `Ordenar-Contar([3],O)`

En todas la llamada a **Ordenar-Contar(L,O)** donde L tiene un solo elemento se devuelve ese elemento (condición de corte). Entonces no se necesita un merge. Por ejemplo en los últimos llamado recursivo **Ordenar-Contar([2],O)** y **Ordenar-Contar([3],O)** van a devolver [2] y [3] respectivamente con 0 inversiones.

A continuación vamos a analizar los mergos en cada uno de los llamados recursivos para el primer participante:

Ordenar-Contar([2,3],O) va a llamar a merge con $A = [4]$ y $B = [3]$ y ambos con 0 inversiones. En **Merge-Contar([4],[3],O)** cuando entre al primer mientras va a compara el orden entre 2 y 3. El orden de 3 es 0 mientras que 4 tiene orden 4 por lo tanto 3 se mete primero en la lista ordenada y se suma a las inversiones la longitud de la primera lista(1) menos el índice de esa lista(0) entonces queda 1 inversión y se le suma 1 al índice de la segunda lista $(j++)$. Ya que $j == |B|$ salimos del primer MIENTRAS y lo que hacemos es meter a la lista final los elementos faltantes (El único elemento de A). Finalizando **Ordenar-Contar([4],[3],O)** va a devolver la lista [3,4] y 1 inversión; **Ordenar-Contar([4,3],O)** va a devolver eso mas el resto de inversiones que eran 0, o sea lo mismo.

Hagamos lo mismo para Ordenar-Contar([1,4,3],O)

```
Ordenar-Contar([1,2,3],O):
 ia,A = Ordenar-Contar([1],O) // = 0,[1]
 ib,B = Ordenar-Contar([2,3],O) // = 1,[3,4]
 i,L =  Merge-Contar([1],[3,4],O)

Merge-Contar(A=[1],B=[3,2],O):
  L = []
  i,j,inv=0
  Primera iteracion (i=0,j=0,inv=0)
    Orden[1] = 1
    Orden[3] = 0
    El orden del elemento de B es menor
        Entonces L = [3], j++ y inv += 1-0 = 1
 Segunda iteracion (i=0,j=1,inv=1)
```

```
    Orden[1] = 1
    Orden[2] = 2
    El orden del elemento de A es menor
      Entonces L = [3,1], i++
  Termina iteracion ya que i == |A|
  Se llena L con los elementos restantes de B // L=[3,1,4]
  Devuelvo inv,L = 1,[3,1,4]
Volviendo a Ordenar-Contar([1,4,3],0)
  vamos a devolver (ia+ib+i,L) = (0+1+1,[3,1,4]) = (2,[3,1,4])
```

Continuamos con Ordenar-Contar([5,2],0)

```
Ordenar-Contar([5,2],0):
 ia,A = Ordenar-Contar([5],0) // = 0,[5]
 ib,B = Ordenar-Contar([2],0) // = 0,[2]
 i,L =  Merge-Contar([5],[2],0)

Merge-Contar(A=[5],B=[2],0):
  L = []
  i,j,inv=0
  Primera iteracion (i=0,j=0,inv=0)
    Orden[5] = 3
    Orden[2] = 2
    El orden del elemento de B es de menor orden
    Entonces L = [2], j++, inv += |A|-i = 1-0
  Termina iteracion ya que j == |B|
  Se llena L con los elementos restantes L=[5,2]
  Devuelvo inv,L = 0,[5,2]
Volviendo a Ordenar-Contar([5,2],0)
  vamos a devolver (ia+ib+i,L) = (0,[5,2])
```

Por ultimo

```
Ordenar-Contar([5,2,1,4,3],0):
 ia,A = Ordenar-Contar([5,2],0) // = 1,[2,5]
 ib,B = Ordenar-Contar([1,4,3],0) // = 2,[3,1,4]
 i,L =  Merge-Contar([2,5],[3,1,4],0)

Merge-Contar(A=[5,2],B=[3,1,4],0):
  L = []
  i,j,inv=0
  Primera iteracion (i=0,j=0,inv=0)
    Orden[2] = 2
    Orden[3] = 0
    El orden del elemento de B es menor
        Entonces L = [3], j++ y inv += |A|-i = 2-0
  Segunda iteracion (i=0,j=1,inv=2)
    Orden[2] = 2
    Orden[1] = 1
    El orden del elemento de B es menor
        Entonces L = [3,1], j++ y inv += |A|-i = 2-0
  Tercera iteracion (i=0,j=2,inv=4)
    Orden[2] = 2
    Orden[4] = 4
    El orden del elemento de A es menor
        Entonces L = [3,1,2], i++
```

```
  Tercera iteracion (i=0,j=2,inv=4)
    Orden[5] = 2
    Orden[4] = 4
    El orden del elemento de A es menor
        Entonces L = [3,1,2,5], i++
  Termina iteracion ya que i == |A|
  Se llena L con los elementos restantes L=[3,1,2,5,4]
  Devuelvo inv,L = 4,[3,1,2,5,4]

Volviendo a Ordenar-Contar([5,4],0)
  va a devolver (ia+ib+i,L) = (1+2+4,[3,1,2,5,4]) = (7,[3,1,2,5,4])
```

Entonces el primer participante tiene 7 inversiones y al ser el primero lo guardamos en el resultado y guardamos sus inversiones como la menor.

3.4.2. Inversiones del resto de los participantes y resultado

Para el segundo participante:

```
Orden = {3:0 , 1:1 , 2:2 , 5:3 , 4:4}
[1,3,2,4,5]
[1,3] [2,4,5]
[1] [3] [2] [4,5]
[1] [3] [2] [4] [5]
[3,1] [2] [5,4]  // 2 Inversiones
[3,1] [2,5,4]
[3,1,2,5,4]
2 inversiones totales
```

El segundo participante tiene menos inversiones por lo tanto lo guardamos en el resultado y ahora la menor cantidad de inversiones es 2. Queda el ultimo participante:

```
Orden = {3:0 , 1:1 , 2:2 , 5:3 , 4:4}
[2, 5, 3, 1, 4]
[2,5] [3,1,4]
[2] [5] [3] [1,4]
[2] [5] [3] [1] [4]
[2,5] [3] [1,4]
[2,5] [3,1,4]
[3,1,2,5,4] // 4 inversiones el 3 y 1 se mueven 2 lugares cada uno
4 inversiones totales
```

Ya que tiene mas inversiones no cambio nada y queda como mejor compañero para el capitán el segundo participantes.

3.5. Código y comparación de complejidad

La solución se encuentra en este repositorio de Github con las pruebas correspondientes. A continuación vamos a compara la solución programada contra el pseudocódigo

3.5.1. Ordenar-Contar

```
def merge_count_order(left:list[int], right:list[int], order:dict[
    int, int]):
     merged = []
     count = i = j = 0
```

```
    while i < len(left) and j < len(right):
        if order[left[i]] <= order[right[j]]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            count += len(left) - i
            j += 1

    merged.extend(left[i:])
    merged.extend(right[j:])

    return merged, count

def merge_sort_count(skills:list[int], order:dict[int, int]):
    if len(skills) <= 1:
        return skills, 0

    mid = len(skills) // 2
    left, left_inv = merge_sort_count(skills[:mid],order)
    right, right_inv = merge_sort_count(skills[mid:],order)
    merged, merge_inv = merge_count_order(left, right, order)

    return merged, (left_inv + right_inv + merge_inv)
```

En este caso el algoritmo es idéntico en cuanto a la complejidad de mergear el problema, la ecuación de recurrencia y su complejidad ($O(nlog(n))$).

3.5.2. Crear-Orden

```
def create_order(captain:Participant):
    order = {}
    for i, skill in enumerate(captain.skills[::-1]):
        order[skill] = i

    return order
```

La primera diferencia que notamos es que se usa una clase **Participante** que tiene nombre y sus habilidades en orden de mayor conocimiento a menor. En cuanto a la complejidad es la misma ya que solo se recorre una vez las habilidades y ingresar elementos a un diccionario cuando todas sus claves son distinta (ilógico tener la misma categoría 2 veces) es constante. Resumiendo **Crear-Orden** y **create_order** tienen la misma complejidad $O(n)$

3.5.3. Buscar-Mejor-Compa

```
def find_participant_with_least_inversions(participants:list[
    Participant],captain:Participant):
    order = create_order(captain)

    min_inversions = float('inf')
    result = None

    for p in participants:
        _, inversions = merge_sort_count(p.skills, order)
```

```
        if inversions < min_inversions:
            min_inversions = inversions
            result = p

    return result
```

Como en el caso de arriba la única diferencia que hay es que se utiliza la clase **Participante**, pero si contar eso la solución en código y en pseudocódigo tienen la misma complejidad de $O(n^2 log(n))$