



Trabajo Práctico N°1

Grupo: Los hijos de Rambo

Fecha de entrega: 25/09/2023

Nombre	Padrón
Joaquin Andresen	102707
Maximiliano Romero	99118
Franco Secchi	106803
Esteban Frascarelli	105965
Tomas Martin	100835

Requisitos	2
Parte 1: La base antártica	3
Proponga y explique una solución del problema mediante Branch and Bound.	3
Pseudocódigo.	3
Realice el análisis de complejidad temporal y espacial.	5
Brinde un ejemplo simple paso a paso del funcionamiento de su solución.	6
Programa su solución.	7
Determine si su programa tiene la misma complejidad que su propuesta teórica.	7
Parte 2: Adivina la carta	8
Determinar y explicar cómo se resolvería este problema utilizando la metodología greedy.	8
De un ejemplo paso a paso. ¿Qué complejidad temporal y espacial tiene la solución?	9
Justifique por qué corresponde su propuesta a la metodología greedy.	9
Parte 3: El concurso	10
Proponer un algoritmo utilizando división y conquista que lo resuelva.	10
Pseudocódigo	10
Analice la complejidad del algoritmo utilizando el teorema maestro y desenrollando la recurrencia.	12
Brindar un ejemplo de funcionamiento.	13
Programa su solución.	15
Analice si la complejidad de su programa es equivalente a la expuesta en el punto 2.	15
Referencias	15

Requisitos

Para poder ejecutar los archivos de la parte 1 y parte 2, hay que dirigirse a sus respectivas carpetas.

Para querer ejecutar la parte 1, primero ejecute el siguiente comando dentro de la carpeta tp1

```
cd antartic_base
```

Luego, ejecute el siguiente comando

```
python antarctic_base.py habilidades.txt candidatos.txt
```

Para querer ejecutar la parte 3, primero ejecute el siguiente comando dentro de la carpeta tp1

```
cd contest
```

Y luego ejecute el siguiente comando

```
python contest.py participantes.txt 4
```

Parte 1: La base antártica

Proponga y explique una solución del problema mediante Branch and Bound.

Para resolver este problema mediante Branch and Bound y Best-first search tuvimos en cuenta los siguientes conceptos

- **Función Costo:** cantidad de habilidades sin cubrir que aporta el candidato

- **Propiedad de corte:** el estado actual tiene más contrataciones que la mejor solución hallada hasta el momento

Partiendo de un nodo con cero contrataciones y vamos a buscar todos los descendientes posibles y por cada uno vamos a calcular la cantidad de habilidades **nuevas** que suman a las contrataciones actuales.

Mientras queden estados disponibles por recorrer, vamos a buscar al candidato que tenga el mejor costo y lo sumamos a nuestra lista de contrataciones actuales.

En caso de que hayamos cubierto todas las habilidades, guardamos el estado actual como la mejor solución.

Pseudocodigo.

```
Obtener_descendientes(nro):
    Sea una lista idx_descendientes vacia
    Sea i desde 0 hasta len(candidatos_nombres) - 1
    Si i es mayor o igual que nro
        agregamos el indice a idx_descendientes
    Retornamos idx_descendientes

Or_binario(lista1, lista2):
    Sea una lista lista_aux vacia
    Sea i desde 0 hasta len(lista1) - 1
        agregamos a lista_aux la operacion OR de la lista1 y lista2
    Retornamos lista_aux

Backtrack(contrataciones_actuales, habilidades_actuales, nro):
    descendientes = Obtener_descendientes(nro)

    Sea una descendientes_costo un diccionario vacio

    Sea idx_descendiente el indice del descendiente de descendientes
    habilidades_ampliadas = Or_binario(habilidades_actuales.COPIAR(),
        candidatos_habilidades[idx_descendiente])
    cant_habilidades_cubiertas = SUMAR(habilidades_ampliadas)
    descendientes_costo[idx_descendiente] = cant_habilidades_cubiertas
    descendientes_no_explorados = len(descendientes)

    Mientras haya descendientes no explorados
        descendientes_no_explorados -= 1
```

```

    idx_mejor_descendiente = buscar el indice del mejor descendiente
    costo_mejor_descendiente = descendientes_costo[idx_mejor_descendiente]
    contrataciones_proximas = contatenamos las contrataciones actuales con la mejor
encontrada
    habilidades_proximas = Or_binario(habilidades_actuales.COPIAR(),
        candidatos_habilidades[idx_mejor_descendiente])

    Eliminados el mejor descendiente del diccionario de costos de descindientes
    Si el costo del mejor descendiente es menor a la suma de las habilidades actuales y la
cantidad de las siguientes contrataciones es menor a la cantidad de la solucion del mejor
candidato:
        Si la suma de las proximas habilidades es igual a la cantidad las proximas
        habilidades mejor_solucion_candidatos = contrataciones_proximas
        mejor_solucion_habilidades = habilidades_proximas
        Retornamos

        Backtrack(contrataciones_proximas, habilidades_proximas, nro + 1)

    Retornamos

Si la cantidad de archivos proporcionados es distinto de 3
    Imprimimos mensaje de error
    Salimos

Sea mejor_solucion_candidatos una lista vacia
Sea mejor_solucion_habilidades una lista vacia
archivo_habilidades = Obtenemos el nombre del archivo de las habilidades
archivo_candidatos = Obtenemos el nombre del archivo de los candidatos

habilidades = Leer_habilidades(archivo_habilidades)
candidatos = Leer_candidatos archivo_candidatos, len(habilidades))
candidatos_nombres = Obtener_claves(candidatos)
candidatos_habilidades = Obtener_valores(candidatos)

mejor_solucion_candidatos = candidatos_nombres
mejor_solucion_habilidades = [1] * len(habilidades)
habilidades_cubiertas = [0] * len(habilidades)
nro = 0
Sea contrataciones una lista vacia
max_candidatos = len(candidatos)
Backtrack(contrataciones, habilidades_cubiertas, nro)
Imprimimos "Mejor solucion:" junto con con la mejor solucion encontrada de los candidatos

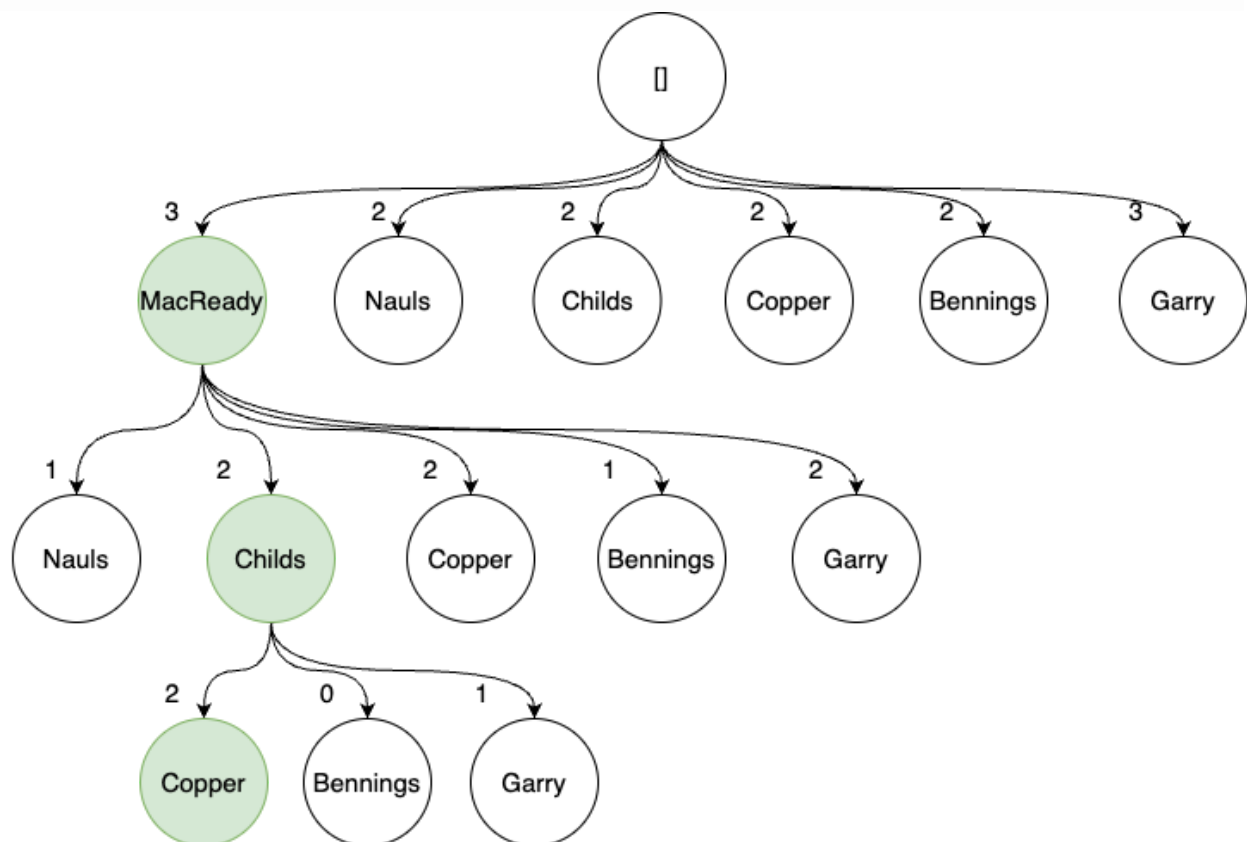
```

Realice el análisis de complejidad temporal y espacial.

El número de candidatos y las habilidades requeridas determinan la complejidad temporal de esta solución. En el peor de los casos, la búsqueda podría explorar todos los subconjuntos de candidatos, lo que daría como resultado una complejidad de $\theta(2^m)$, donde m es el número de candidatos.

La complejidad espacial depende del espacio utilizado para la recursión, así como del espacio requerido para almacenar los datos y las habilidades de los candidatos. Debido a que la profundidad máxima de la recursión es m , la complejidad espacial es $\theta(m)$.

Brinde un ejemplo simple paso a paso del funcionamiento de su solución.



En este diagrama se puede observar cómo se comportaría el algoritmo.

Primero busca todos los candidatos posibles y calcula el costo. En este caso tenemos dos candidatos que suman 3 habilidades sin cubrir, pero por ser el primero en la lista elegimos a MacReady.

Sumamos a MacReady a nuestra lista de contrataciones y repetimos.

En este caso, el Childs es el primero de los 3 candidatos que suman más habilidades. Entonces lo sumamos a nuestra lista de contrataciones, pero como todavía no llegamos a cubrir las 7 habilidades, repetimos el proceso.

Con Copper, logramos cubrir las 7 habilidades. Entonces guardamos la lista como la mejor solución y arrancamos de nuevo empezando con Garry para ver si hay otra mejor combinación de candidatos.

La diferencia entre la primera la pasada y las siguientes, es que vamos a realizar una poda cuando tengamos más de tres candidatos en nuestra lista.

En este caso particular tuvimos la suerte de hallar la solución óptima en la primera pasada debido a la ubicación de los candidatos en la lista.

Programe su solución.

Para poder ejecutar la solución, es necesario ingresar a la carpeta [tp1/antartic_base](#). La misma puede encontrarse dentro de la carpeta [tp1/antartic_base](#) de la solución enviada.

Determine si su programa tiene la misma complejidad que su propuesta teórica.

Se estimó en la propuesta teórica que la complejidad temporal es de $\Theta(2^m)$ y la complejidad espacial es de $\Theta(m)$.

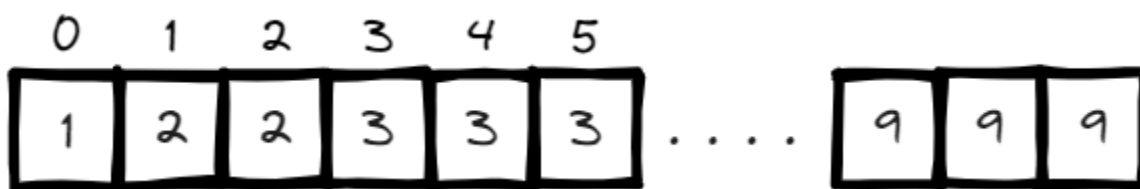
Luego del análisis realizado en el ítem anterior, observamos en nuestro programa que la complejidad temporal, por el número de candidatos y habilidades, es de $\Theta(2^m)$. Después, se descubrió que $\Theta(m)$ era la complejidad espacial.

Por lo tanto, podemos deducir que nuestro programa es tan complejo como la propuesta teórica.

Parte 2: Adivina la carta

Determinar y explicar cómo se resolvería este problema utilizando la metodología greedy.

El problema consiste en un 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". Podemos pensar que el mazo de cartas (ordenado) es un array de números que sigue el siguiente formato.

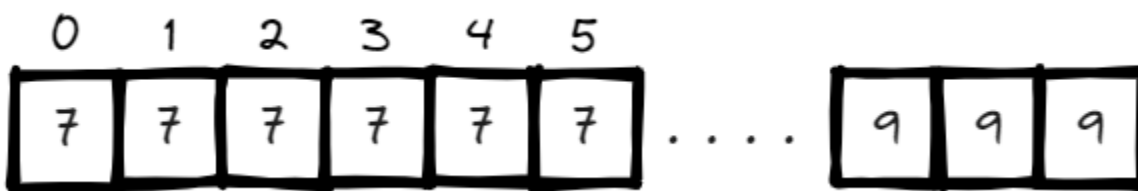


Ahora bien, el siguiente paso en este problema es mezclar las cartas y repartir una carta al rival para luego adivinar que carta es la que tiene en la mano.

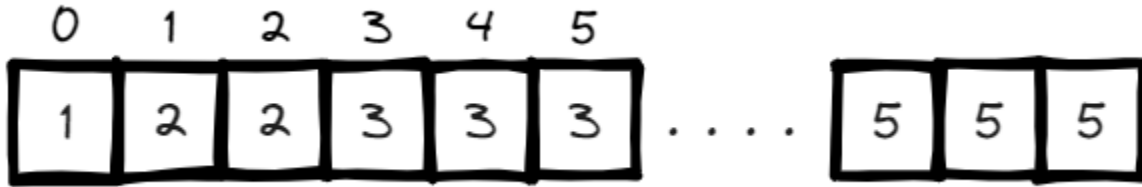
La resolución de este problema mediante una metodología Greedy viene dada por el hecho de que vamos a "dividir el problema en subproblemas" con el objetivo de solucionar problemas de forma local para llegar a la solución global, recordemos que un algoritmo Greedy es aquel que no mira la situación global, sino que elige localmente lo que considera mejor según un criterio preestablecido.

Aquí al momento de hacerle preguntas al rival, estaremos dividiendo el array en mitades y quedándonos con la mitad correspondiente, originando un nuevo subproblema sobre el que aplicaremos el mismo criterio.

Esto que acabamos de decir, lo detallaremos mejor en el próximo punto, pero la primera pregunta que le haremos es "¿Es el 6 el número elegido?", en el caso de que la respuesta sea "No", preguntaremos si el número es mayor a 6. Si la respuesta es "Sí" nos quedaremos con el siguiente array:



Si la respuesta es "No" nos quedaremos con la otra mitad del array



Y luego, volveremos a repetir el procedimiento.

De un ejemplo paso a paso. ¿Qué complejidad temporal y espacial tiene la solución?

Para determinar cómo subdividir el problema en este caso, utilizaremos las respuestas "si o no" de nuestro rival.

Primero ubicamos el número 6 en la mitad del array y preguntaremos: "¿Es el 6 el número elegido?"

En el caso de no ser así, preguntaremos si el número elegido es mayor o menor que el 6, y si es mayor, entonces nos quedaremos con esa mitad del array en lugar de la primera mitad. Por lo tanto, reduciremos el problema hasta obtener el número.

Dado que no será necesario recorrer todo el array y siempre lo dividiremos en partes similares, la solución tendrá una complejidad temporal de $\Theta(\log(n))$.

Será de n debido a la complejidad espacial, ya que tendremos que cargar todo el arreglo en memoria y luego mover los índices para obtener cada subdivisión del array.

Justifique por qué corresponde su propuesta a la metodología greedy.

Corresponde a una metodología Greedy porque estamos dividiendo el problema global en pequeños subproblemas y solucionando cada uno de forma local siguiendo un mismo criterio. Esto mismo está también desarrollado en la primera pregunta.

Parte 3: El concurso

Proponer un algoritmo utilizando división y conquista que lo resuelva.

En primer lugar, tuvimos que establecer cómo podríamos encontrar al participante que mejor se complementará con el capitán seleccionado en función de los conocimientos de ambos. Para ello, consideramos la posibilidad de utilizar el Conteo de Inversiones. En este método, dos concursantes serán más complementarios cuanto mayor sea la cantidad de inversiones que deban realizarse para que el listado de categorías quede exactamente en la misma posición. Esto nos llevó a concluir que no se trata de un Conteo de Inversiones tradicional, en el que se ordena según el orden creciente de los números, sino que debe ordenarse de acuerdo con las categorías del capitán.

Nuestro algoritmo de Conteo de Inversiones se implementó mediante la técnica de división y conquista. En este proceso, el listado de un participante se divide en dos partes, y así sucesivamente, hasta que solo queda un elemento en cada una. En ese punto, comparamos estos dos valores (número de categoría) y realizamos una inversión en caso de que el capitán tenga un mayor conocimiento en la categoría de la derecha que en la de la izquierda. Luego, obtenemos dos sublistados, cada uno con dos elementos, y generamos un listado ordenado nuevamente en función de los conocimientos del capitán. Este proceso se repite hasta que el listado del participante quede exactamente ordenado como el del capitán elegido.

Una vez que conocemos la cantidad de inversiones necesarias de cada participante con respecto al capitán, elegimos como compañero al que tenga el mayor valor.

Pseudocódigo

```
Sea C el capitán. Con su nombre y listado de categorías
Sea P los demás participantes. Con su nombre y listado de categorías
Sea M la cantidad de categorías
```

```
CalcularInversiones(A, B, Cat_Capitan):
```

```
    Sea X el listado ordenado
```

```
    Cant_Inver = 0
```

```
    Mientras A y B tengan elementos:
```

```
        Sea a el primer elemento de A
```

```
        Sea b el primer elemento de B
```

```
        Si a esta primero dentro de Cat_Capitan con respecto a b:
```

```
            Inserto a en X
```

```

        Elimino a de A
    Sino:
        Inserto b en X
        Elimino b de B
        Cant_Inver += longitud de A

Mientras A tenga elementos:
    Inserto todos los elementos restantes de A en X

Mientras B tenga elementos:
    Inserto todos los elementos restantes de B en X

Retorno X, Cant_Inver

ContarInversiones(M, Cat_Capitan):
    Categorias = longitud de M

    Si Categorias es 1:
        Retorno M, 0

    Mitad = Categorias // 2
    Izquierda = M[:mitad]
    Derecha = M[mitad:]

    Ordenado_Izq, Inver_Izq = ContarInversiones(Izquierda, Cat_Capitan)
    Ordenado_Der, Inver_Der = ContarInversiones(Derecha, Cat_Capitan)
    Final, Cant_Inversiones = CalcularInversiones(Ordenado_Izq, Ordenado_Der,
Cat_Capitan)
    Cant_Inversiones += Inver_Izq + Inver_Der

    Retorno Final, Cant_Inversiones

Nombre_Comp = "" // Nombre del mejor compañero hasta el momento
Max_Inv = -1 // Cantidad de inversiones del mejor compañero hasta el momento
Por cada m dentro de M:
    _, Cant_Inversiones = ContarInversiones(m['categorias'], Cat_Capitan)
    Si Cant_Inversiones > Max_Inv:
        Max_Inv = Cant_Inversiones
        Nombre_Comp = m['nombre']

```

Analice la complejidad del algoritmo utilizando el teorema maestro y desenrollando la recurrencia.

El teorema maestro se puede aplicar dentro de la función *ContarInversiones*. En esta función, podemos observar que el problema se divide en dos subproblemas, cada uno con $m/2$ elementos (siendo m la cantidad de categorías), y en el caso base, la complejidad es $\Theta(1)$.

Por otro lado, la fusión entre los dos subproblemas se realiza dentro de la función *CalcularInversiones*, lo que tiene un tiempo de ejecución de $\Theta(m)$ debido a que la comparación entre dos elementos puede resolverse en $\Theta(1)$.

Planteando el Teorema Maestro: $T(m) = aT(\frac{m}{b}) + f(m)$; con $a \geq 1$, $b \geq 1$ y $T(0)$ cte

En nuestro caso tenemos que:

- $a = 2$.
- $b = 2$.
- $f(m) = O(m)$.

Por lo que:

- $T(m) = 2T(\frac{m}{2}) + O(m)$
- $T(1) = O(1)$

Para ver la complejidad de nuestro algoritmo vamos a probar primero el caso 2 del Teorema Maestro

Caso 2

Debemos ver: $f(m) = \Theta(m^{\log_b a}) \Rightarrow T(m) = \Theta(m^{\log_b a} \cdot \log m)$.

Entonces: $m = \Theta(m^{\log_2 2}) = \Theta(m^{\log_b a}) = \Theta(m)$.

Como se cumple, podemos determinar que nuestra complejidad de la función *ContarInversiones* es $\Theta(m \cdot \log m)$:

$$\Theta(m^{\log_b a} \cdot \log m) = \Theta(m^{\log_2 2} \cdot \log m) = \Theta(m \cdot \log m)$$

Como esa función se debe calcular para cada una de los participantes que no es el capitán, nos quedaría que todo nuestro algoritmo que resuelve el problema será (siendo n la cantidad de participantes):

$$\Theta(n \cdot [m \cdot \log m])$$

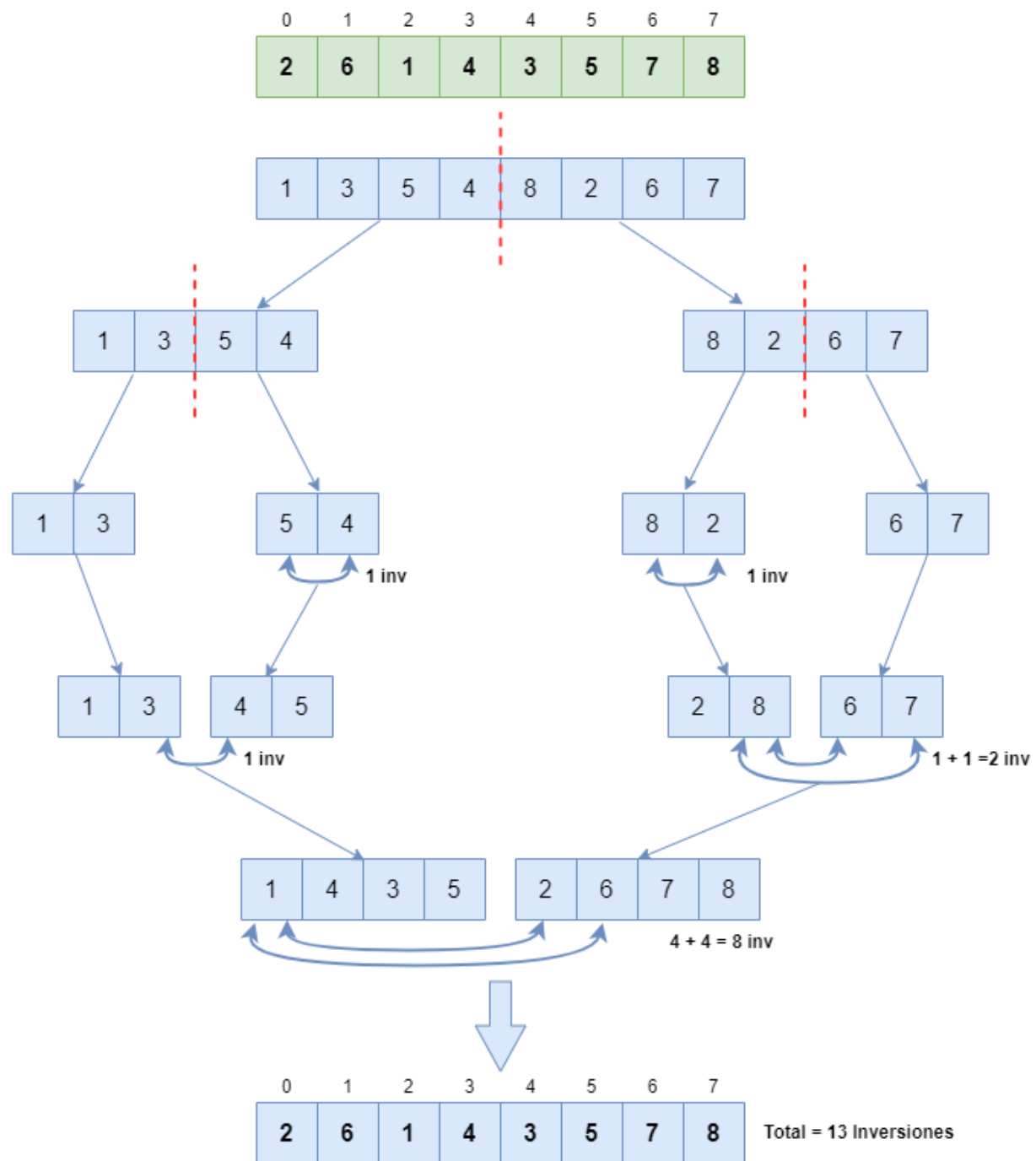
Brindar un ejemplo de funcionamiento.

Si elegimos que nuestro capitán tenga el siguiente conocimiento: [2, 6, 1, 4, 3, 5, 7, 8]

Y por otro lado, si el concursante el cual queremos ver que tan bien es complementario con el capitán posee conocimientos en: [1, 3, 5, 4, 8, 2, 6, 7]

En el siguiente diagrama se puede observar el funcionamiento del algoritmo planteado. En verde se encuentra el listado de categorías del capitán (vector objetivo) y en azul el listado de nuestro participante.

En cada uno de los pasos se muestra la partición de cada subproblema y la cantidad de inversiones en cada ocasión



Programe su solución.

La misma puede encontrarse dentro de la carpeta **tp1/contest** de la solución enviada.

Analice si la complejidad de su programa es equivalente a la expuesta en el punto 2.

Al correr el programa dentro del archivo **contest.py** podemos inferir lo siguiente de la complejidad de cada función que se llama:

Siendo n la cantidad de concursantes, y m la cantidad de categorías.

- **leer_participantes**: se encarga de leer los tanto los participantes con su listado de categorías como del capitán, del archivo pasado como parámetro. Tiene una complejidad de $\Theta(n \cdot m)$
- **crear_diccionario_capitan**: crea el diccionario en donde las claves son las diferentes categorías y su valor es el índice en donde se encuentra en el listado de categorías del capitán. Tiene una complejidad de $\Theta(m)$
- **obtener_cantidad_inversiones**: calcula la cantidad de inversiones que deben realizarse entre un participante y su capitán. Esto se debe ejecutar una cantidad de n veces.



Dentro de este método se utiliza División y Conquista de la misma forma que se diagramó en el pseudocódigo del primer punto.

Por otro lado, el merge de las inversiones también se ajustan a lo que habíamos pensado, ya que al utilizar un diccionario de python, el indexado por la clave, que sería el número de categorías, se resuelve en $\Theta(1)$

Por lo que nuestro algoritmo en Python también tiene una complejidad temporal de:

$$\Theta(n \cdot [m \cdot \log m])$$

Referencias

- Branch & Bound:  Branch & Bound - Introduccion.pdf
- Greedy:
<https://docs.google.com/document/d/1NO1DQJmM-Ym1CZzsSiok03wLChCOV6ArelMt0X-tXfM>
- División y Conquista:  Apunte TDA - División y Conquista
- Documentación Python: <https://docs.python.org/es/3.11/>