



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	3
Parte 1: La base antártica	3
Proponga y explique una solución del problema mediante Branch and Bound.	3
Pseudocódigo.	4
Realice el análisis de complejidad temporal y espacial.	6
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
Anexo: Correcciones	16
Parte 1: La base antártica	16
Pseudocódigo corregido	17
El pseudocódigo corresponde mas a Depth-first branch-and-bound que Best First	18
La función sumar no la detalla. ¿Qué complejidad tiene? para buscar la opción que mas cubre entre los candidatos disponibles. como lo realiza? realiza cada vez una búsqueda entre los disponibles? o hace un ordenamiento? que complejidad tiene este paso?	18
Complejidad	21
Parte 2: Adivina la carta	23
Determinar y explicar cómo se resolvería este problema utilizando la metodología greedy.	23
De un ejemplo paso a paso. ¿Qué complejidad temporal y espacial tiene la solución?	23
Justifique por qué corresponde su propuesta a la metodología greedy.	24



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 o igual 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

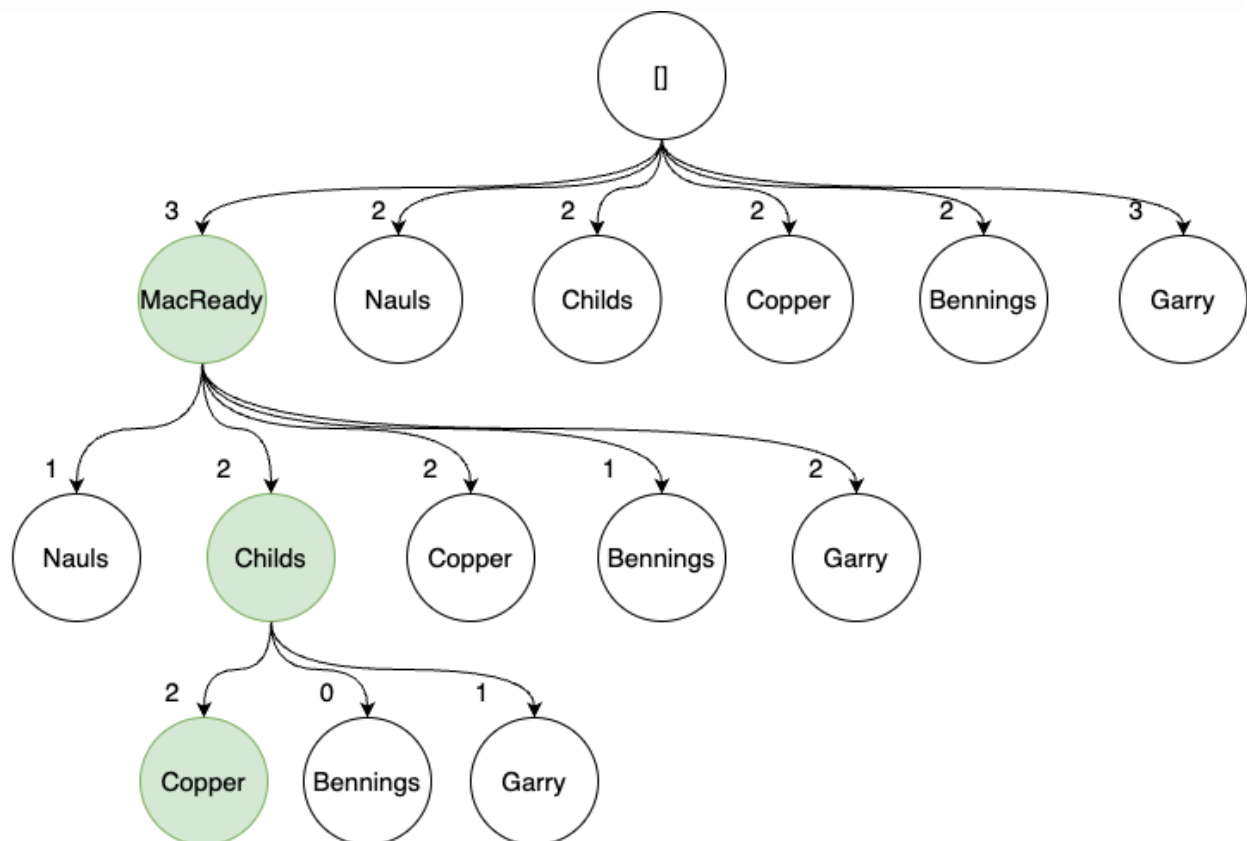
```

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 $\Theta(m)$.

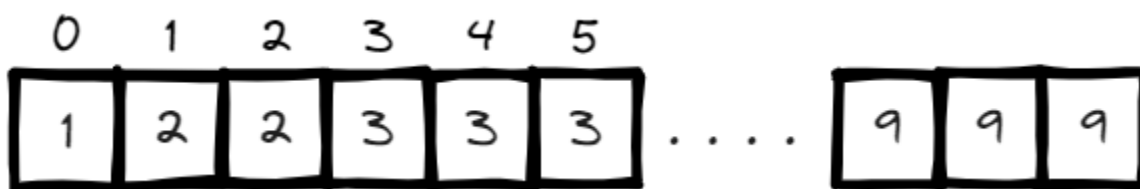
Luego del análisis realizado en el ítem anterior, observamos en nuestro programa que la complejidad temporal, por el número de candidatos y habilidades, es de $\Theta(2^m)$. Después, se descubrió que $\Theta(m)$ era la complejidad espacial.

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.

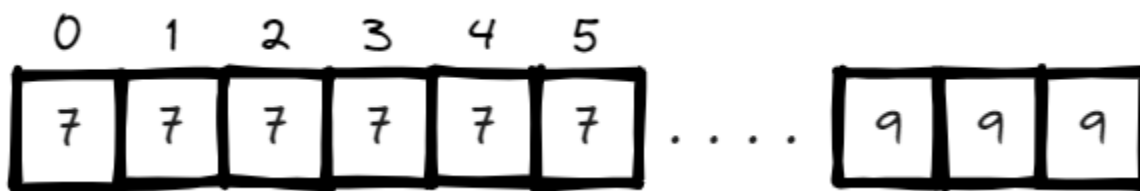


Ahora bien, el siguiente paso en este problema es mezclar las cartas y repartir una carta al rival para luego adivinar que carta es la que tiene en la mano.

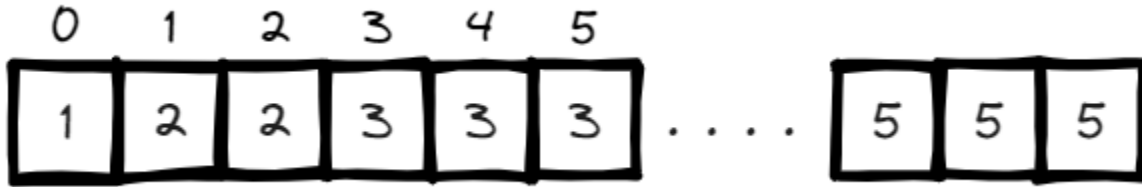
La resolución de este problema mediante una metodología Greedy viene dada por el hecho de que vamos a "dividir el problema en subproblemas" con el objetivo de solucionar problemas de forma local para llegar a la solución global, recordemos que un algoritmo Greedy es aquel que no mira la situación global, 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):

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

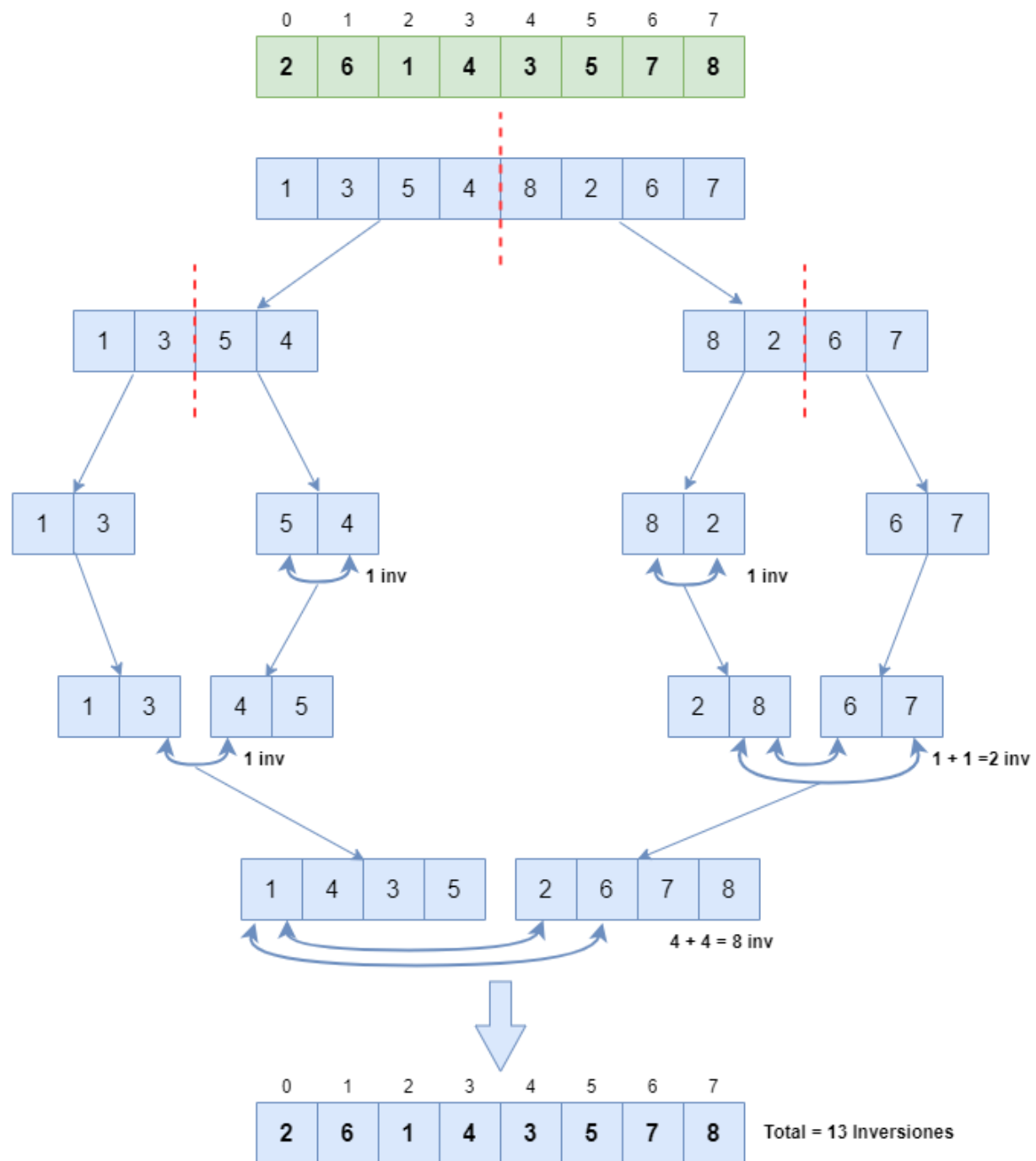
Brindar un ejemplo de funcionamiento.

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/>

Anexo: Correcciones

Parte 1: La base antártica

Pseudocódigo corregido

```
Obtener_descendientes(contrataciones_actuales):
    Sea candidatos_nombres los nombres de todo el personal
    Retornar aquellos candidatos_nombres que no se encuentren en contrataciones_actuales

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):
    descendientes = Obtener_descendientes(contrataciones_actuales)

    Sea una descendientes_costo un diccionario vacio

    Por cada nombre_descendiente dentro de descendientes
        habilidades_ampliadas = Or_binario(habilidades_actuales,
            candidatos_habilidades[nombre_descendiente])
        cant_habilidades_cubiertas = sum(habilidades_ampliadas)
        descendientes_costo[nombre_descendiente] = cant_habilidades_cubiertas

    Mientras haya descendientes no explorados
        descendientes_no_explorados -= 1
        nombre_mejor_descendiente = busco nombre del descendiente con el mejor costo
        costo_mejor_descendiente = descendientes_costo[nombre_mejor_descendiente]
        contrataciones_proximas = contatenamos las contrataciones actuales con la mejor
encontrada
        habilidades_proximas = Or_binario(habilidades_actuales,
            candidatos_habilidades[idx_mejor_descendiente])

        Eliminados el mejor descendiente del diccionario de costos de descendientes

        Si el costo del mejor descendiente es mayor a la suma de las habilidades actuales y la
cantidad de las siguientes contrataciones es menor a la cantidad de la mejor solucion
hallada:
            Si se cubrieron todas las 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)
max_candidatos = len(candidatos)
Sea contrataciones_inicial una lista vacia
hab_cubiertas_inicial = [0] * len(habilidades)

Backtrack(contrataciones, hab_cubiertas_inicial)

Si mejor_solucion_candidatos vacio:
    Imprimimos que los candidatos no cubren todas las habilidades
Sino:
    Mostramos mejor_solucion_candidatos

```

El pseudocódigo corresponde mas a Depth-first branch-and-bound que Best First
Efectivamente el código finalmente utilizado corresponde a Depth-First Branch-and-Bound.

No queda claro porque hace un "habilidades_actuales.COPIAR()" antes de llamar a la función "OR_Binario" si dentro de esta no modifica a "habilidades_actuales"

Quitar el copy, no es necesario ya que adentro se genera una nueva lista auxiliar.

La función sumar no la detalla. Qué complejidad tiene? para buscar la opción que mas cubre entre los candidatos disponibles. como lo realiza? realiza cada vez una búsqueda entre los disponibles? o hace un ordenamiento? que complejidad tiene este paso?

La función *sumar* se utiliza para calcular la suma del vector resultante en el *or_binario* entre las habilidades actuales y las del posible descendiente a elegir. Esto representaría la cantidad de habilidades que se cubrirán en caso de seleccionar a dicho candidato. La misma tiene una complejidad temporal de $O(n)$ siendo n la cantidad de habilidades a cubrir, y una espacial de $O(1)$.

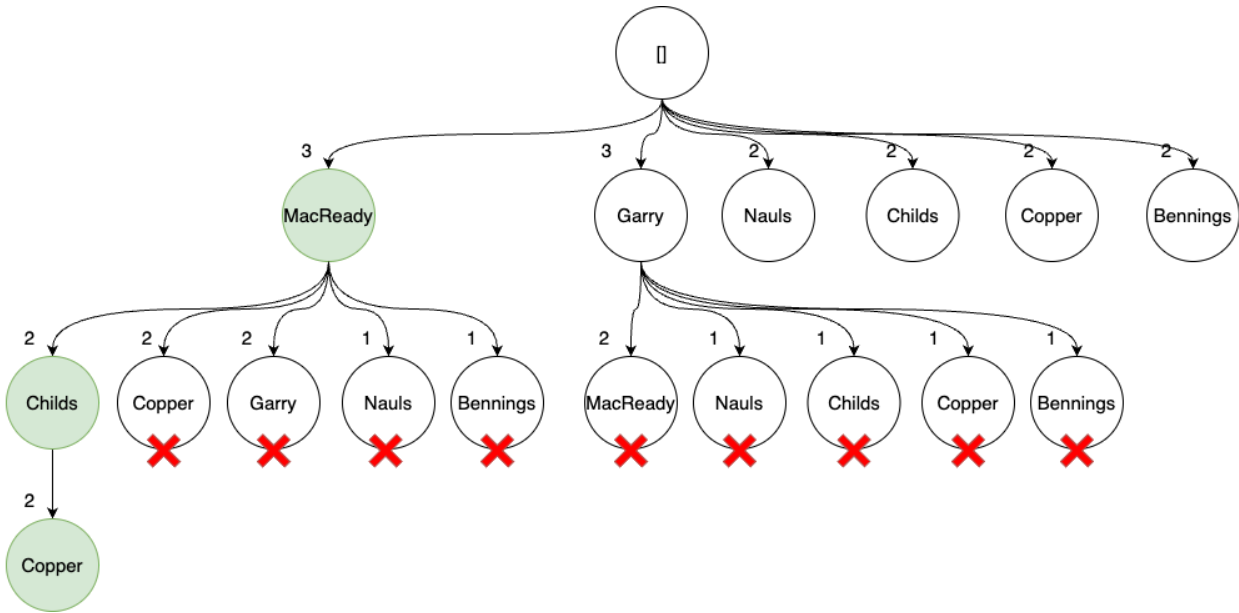
Para buscar la opción que más cubre, obtenemos el índice que corresponde al mayor valor del vector de habilidades cubiertas (cada índice, corresponde a un candidato). En cada vuelta realizar una búsqueda entre los candidatos disponibles, por lo que, al no tener un orden, la complejidad de buscar el índice que corresponde al mayor valor del vector de habilidades cubiertas es $O(c)$ siendo "c" la cantidad de candidatos disponibles.

Intento comprender esta línea de su pseudocodigo "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 solución del mejor candidato:". Se define el costo como "cantidad de habilidades sin cubrir que aporta el candidato". Que seria " la cantidad de las siguientes contrataciones"??? seria "contrataciones_proximas"? No es claro que esta haciendo ahí. no es claro el pseudocodigo en ese punto.

En nuestro pseudocódigo el costo hace referencia a la cantidad de habilidades que quedarían cubiertas al seleccionar a un candidato dado. Es decir que si hasta el momento se tenían cubiertas 5 habilidades, y dicho candidato agrega una más que no se está abarcando, el costo de seleccionarlo sería de 6. Mientras que si no agrega ninguna nueva, el costo sería de 5. Mediante esa línea querríamos explicar que si la cantidad de habilidades que aporta el candidato actual (el que tenía el mayor costo entre los posibles nodos del árbol) da como resultado una cantidad de habilidades mayor a las que teníamos cubiertas hasta el momento y además, en menos cantidad de contrataciones, sabemos que es viable explorar esta rama ya que nos está dando una solución mejor de la que teníamos.

La variable *contrataciones_proximas* representa las contrataciones en caso de sumar al candidato a la solución actual.

El ejemplo paso a paso que brinda no termina de ser del todo claro, ni completo. Dice "arrancamos de nuevo empezando con Garry para ver si hay otra mejor combinación de candidatos.". Cual de todos los "Garry"? En su grafo hay 3. En su ejemplo no se muestra ninguna poda. solo dice que encontró en la primera pasado el optimo. podría detallar mas? (hacer mas graficos)



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.

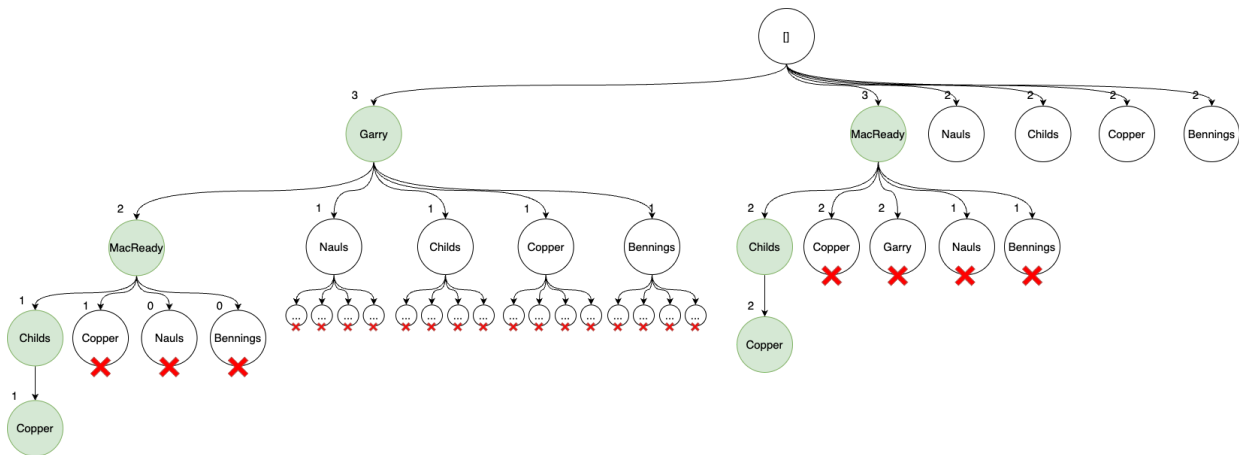
Con Copper, logramos cubrir las 7 habilidades. Entonces guardamos la lista como la mejor solución. A partir de ahora la poda se va a realizar cuando lleguemos a 3 contrataciones.

El algoritmo continúa explorando el resto de los descendientes disponibles de MacReady pero ninguno alcanza para cubrir todas las contrataciones y sumar descendientes nos llevará a superar la propiedad de corte.

Luego de explorar todos los descendientes de MacReady, seguiremos por los de Garry, donde no encontraremos ninguna solución mejor, y pasara lo mismo con los de Nails, Childs, Cooper y Bennings.

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, y por esa razón siempre vamos a cortar luego de comprobar que dos candidatos no alcanzan a completar todas las habilidades.

Ahora vamos a ver otro caso en el cual no tenemos la “suerte” de encontrarnos con la solución óptima en la primera pasada. Para ello intercambiamos las posiciones de MacReady y Garry en el archivo candidatos.txt.



En este caso podemos ver que la primera solución que hallamos tiene 4 candidatos (Garry, MacReady, Childs y Copper). Eso hará que la poda se produzca cuando lleguemos a 3 candidatos.

Pero cuando llegamos a la rama principal de MacReady, vamos a encontrar una solución mejor que cambiará en que momento se realizará una poda.

Complejidad

Al correr el programa dentro del archivo `antarctic_base.py` podemos ver que la mayor complejidad va a estar impuesta por el método `backtrack`.

Desmenuzando dicha función inferir lo siguiente de la complejidad de cada método que utiliza: Siendo n la cantidad de candidatos, y m la cantidad de habilidades a cubrir.

- **obtener_descendientes**: se recorren todos los nombres de posibles candidatos por lo que se van a recorrer $O(n)$, luego la verificación de pertenencia dentro de la lista de las contrataciones actuales tiene un orden de $O(1)$. Por lo tanto la complejidad temporal final es de $O(n)$. La lista de nombres de candidatos se almacena en memoria y la utiliza la función de backtrack, entonces su complejidad espacial es de $O(n)$
- Luego por cada descendiente (posible candidato) se realiza:
 - **Or_binario**: se recorren la lista de habilidades actuales y la que posee cada posible candidato en busca de obtener las habilidades finales en caso de seleccionar al candidato. Entonces se itera a través de las m habilidades de las listas y realiza operaciones de acceso y combinación en tiempo constante para cada elemento. Lo que indica que la complejidad temporal es de $O(m)$.
 - **sum**: se utiliza para contar todas las habilidades que se cubrirán en caso de seleccionar al candidato. Apoyándonos en la documentación oficial de python,

que describe que para esta operación se recorre cada elemento de la lista, podemos determinar que su complejidad temporal es de $O(m)$ y complejidad espacial de $O(1)$

El resultado de todo este bucle se almacena en un diccionario en donde la clave es cada posible candidato (n) y el valor es un número (el resultado de la función *sum*).

Como este proceso se realiza por cada candidato tenemos una complejidad temporal de $O(n \cdot m)$ y una espacial de $O(n)$.

- Luego de obtener la cantidad de habilidades que aportaría en caso de seleccionarlo, por cada uno de ellos se realiza lo siguiente:
 - **max**: la operación de encontrar la clave con el valor máximo implica iterar a través de todas las claves en el diccionario y comparar los valores asociados con esas claves. Por lo tanto, la complejidad temporal de esta expresión será $O(n)$ y el resultado es un string por lo que la complejidad espacial es de $O(1)$
 - Obtenido el nombre del mejor candidato, se obtiene la cantidad de habilidades, y al contar con un diccionario indexado por el nombre del candidato, el acceso es de $O(1)$
 - Se almacena el nombre de los candidatos seleccionados hasta el momento y el nombre del mejor candidato entonces la complejidad espacial es de $O(n)$. Y temporal de $O(1)$ ya que solo se inserta el nombre del “nuevo” candidato seleccionado y esta operación tiene un costo constante
 - **Or_binario**: como se explicó anteriormente tiene una complejidad temporal de $O(m)$
 - **del**: la eliminación de una clave de un diccionario tiene un coste de $O(1)$ dada la documentación oficial del lenguaje utilizado
 - Evaluación de condición: en esta parte se realizan las evaluaciones de si el candidato efectivamente aporta una mejora en las habilidades cubiertas hasta el momento en la rama actual, además de verificar que no sea peor que la mejor solución encontrada hasta el momento y también si no se están satisfaciendo hasta el momento todas las habilidades.

Las operaciones que se realizan son **sum**, que como ya vimos antes tiene una complejidad temporal de $O(m)$. Y de **len**, que es una función propia del lenguaje utilizado en la programación que tiene un costo de $O(1)$.

Por lo que la complejidad sería de $O(n \cdot [n + m])$

Por lo tanto la complejidad temporal de nuestra solución está principalmente establecida por explorar todas las combinaciones posibles en el peor de los casos, que tendría un orden de $O(n!)$. Como en cada subproblema tenemos una complejidad de $O(n \cdot [n + m])$, la complejidad temporal será de $O(n! \cdot n \cdot [n + m]) \approx O(n! \cdot n^2)$.

Mientras que cada nodo de nuestra solución requiere de $O(n)$ para mantener los posibles descendientes y como se deben recorrer todas las combinaciones ($n!$), tendremos una complejidad espacial de $O(n! \cdot n)$.

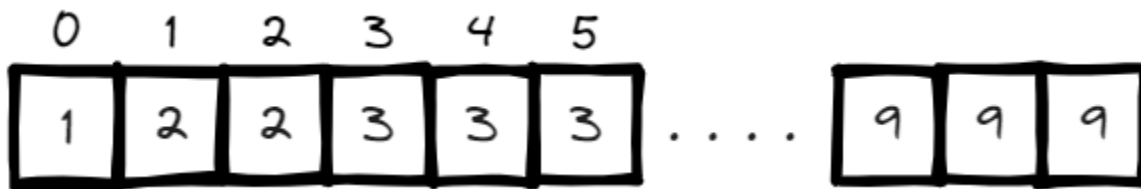
Por otro lado, el análisis de la complejidad de nuestra solución corresponde también al pseudocódigo propuesto.

Parte 2: Adivina la carta

Al momento de corregir el problema encontramos que hay una forma más óptima de resolver el problema y es utilizando el método de Huffman. Para poder explicar mejor en qué consiste, explicaremos los mismos puntos que explicamos anteriormente.

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.

Para resolver dicho problema, planteamos la solución con el método de Huffman.

El método de Huffman consiste en un algoritmo que permite codificar mensajes de forma Greedy, basándose principalmente en la frecuencia de un mensaje dentro de su fuente. Esto logra que mensajes más frecuentes tengan longitudes de código más pequeñas.

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

Lo primero que haremos será crear 9 nodos de un árbol y cada nodo tendrá una propiedad llamada *peso*. El peso será equivalente a la cantidad de repeticiones de cada elemento dentro de la fuente, esto quiere decir que el nodo 1 tendrá peso 1, el nodo 2 tendrá peso 2 y así con cada uno.

Luego insertamos estos nodos en un Heap de mínimos, lo que dejará en el tope del heap al elemento con el menor peso. Con el Heap construido, haremos un programa iterativo que hará lo siguiente:

1. Obtener los 2 primeros elementos del heap
2. Se creará un nuevo nodo "x" que será el padre de dichos elementos y su peso será la suma de ambos pesos
3. Finalmente este elemento se insertará en el Heap

Esta iteración finalizará cuando quede un solo elemento en el Heap y dicho elemento será la raíz del árbol.

Finalmente cada hoja del árbol será un elemento de la fuente y su código estará generado mediante la siguiente forma:

$$Longitud(C(W)) = \sum w_i * size(C_i)$$

Siendo $C(W)$ los códigos prefijos y binarios, w_i el peso del nodo i y $size(C_i)$ la longitud de la codificación.

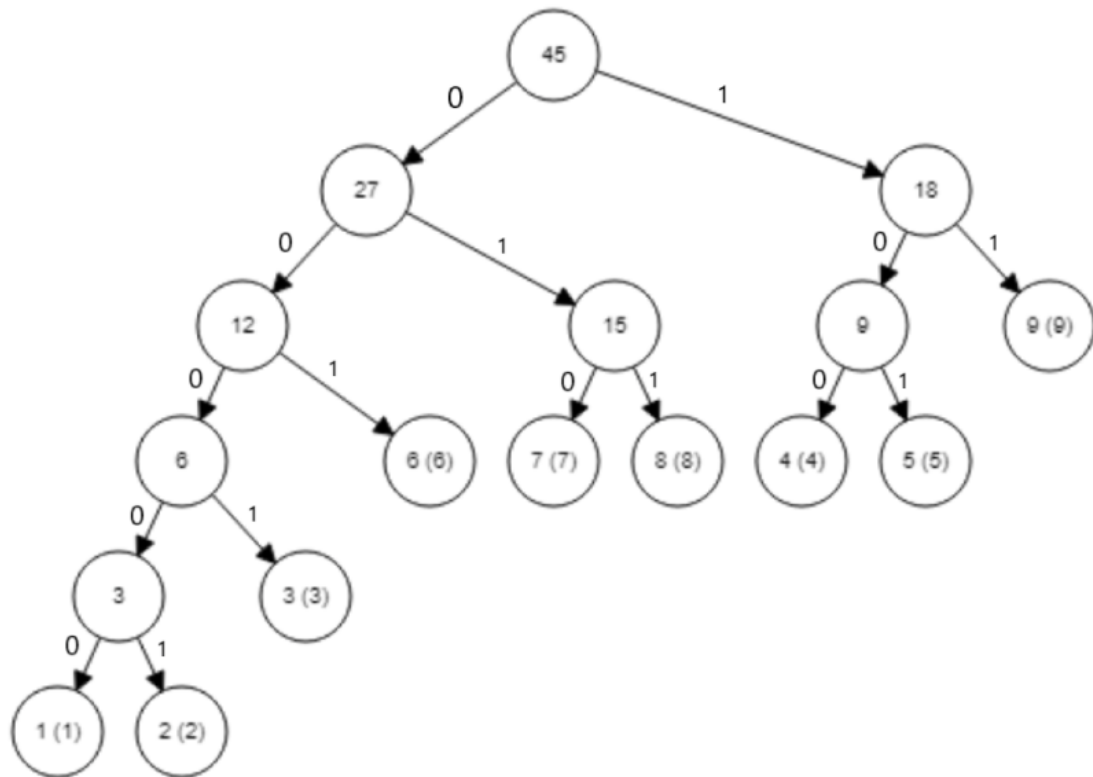
Esto generará que cada código tenga la mínima una longitud de codificación, de cualquier combinación posible de caracteres, y dando un acceso más rápido al elemento con mayor frecuencia dado por la fuente.

Al aplicar este algoritmo a nuestro ejercicio, recordemos que cada elemento de nuestro tendrá un peso i , que será su código, ya que tendremos un 1, 2 dos, etc.

Sabiendo esto crearemos un heap de mínimos. En él incluiremos nodos cuyo caracter y peso será i . De modo que en este Heap tendrá 9 elementos, con el nodo 9 en el final del heap y el nodo 1 en su comienzo.

Lo que haremos ahora será iterar hasta dejar un elemento en el heap. Dentro de la iteración el primer paso es crear un **nodo z**, que tendrá como hijos los dos primeros nodos desapilados. Es decir que en este caso el primer **nodo z** tendrá como hijos al nodo 1 y nodo 2, y el peso del nodo z, será la suma del peso de ambos nodos, el **nodo z** tendrá peso 3. Finalmente se insertará al **nodo z** en el heap, y el ciclo comenzará de nuevo.

Este proceso seguirá hasta que el el heap tenga solo un elemento, lo que nos dejaría el último nodo de la siguiente manera:



Dicho nodo será la raíz del árbol, y eso será lo que retornemos. De esta forma el nodo 1, tendrá como código al "00000", y el código para el 9 será "11".

Por cada símbolo codificado hay que recorrer el árbol para decodificarlo. El árbol contiene n nodos y, por término medio, se necesitan $\log(n)$ visitas a nodos para decodificar un símbolo. Por tanto, la complejidad temporal sería $\Theta(n \log(n))$ siendo n la cantidad de elementos de la fuente.

La complejidad espacial del algoritmo de Huffman depende de la implementación del mismo, en nuestro caso, la complejidad es de $\Theta(n)$, siendo n la cantidad de símbolos únicos de la fuente. Ya que serán $k * n$ nodos para los árboles, siendo k una constante que será dada por la cantidad de "nodos z" que se creen para generar el árbol, y sabemos que una complejidad $k * n$, será de complejidad n .

Justifique por qué corresponde su propuesta a la metodología greedy.

Nuestra propuesta es Greedy porque se toma una decisión óptima en cada paso sin considerar el impacto de esa decisión en los pasos siguientes. El algoritmo de Huffman comienza con un árbol de símbolos de la fuente de entrada, donde los símbolos que se pueden extraer se representan en nodos y los símbolos de los hijos de cada nodo se concatenan. El algoritmo elige qué dos nodos combinar de forma localmente óptima en cada paso, sin tener en cuenta el impacto de esa elección en el árbol final.

