UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE INGENIERÍA
Año 2022- 1er Cuatrimestre

[75.29/95.06] TEORÍA DE ALGORITMOS

Trabajo Práctico Nº2: "División y Conquista, y Computación Dinámica"

Padrón	*Apellido y Nombre*	*Email*
93310	Cabrera, Jorge	jocabrera@fi.uba.ar
93042	Anez, Johana	janez@fi.uba.ar
92354	Serra, Diego	dserra@fi.uba.ar
107587	Gallino, Pedro	pgallino@fi.uba.ar
104126	Lamanna, Tobias	tlamanna@fi.uba.ar

Fecha 1er entrega: 27/04/2022

Código fuente:

Grupo N° 2

Parte 1

1. Explicar cómo se puede resolver este problema por fuerza bruta. Analizar complejidad espacial y temporal de esta solución.

Resolver el problema mediante fuerza bruta resulta muy sencillo. Basta con tomar cada jugador del ranking y analizar cuantos jugadores superó. Bajándolo a código, se tomaría cada jugador en una iteración sobre todos ellos, y con cada uno se haría otra iteración, sobre los demás jugadores, verificando si lo superó o no.

pseudocódigo:

```
resolucion_fuerza_bruta(jugadores):
    superaciones = {}
    for jugador in jugadores:
        for rival in rivales:
            si jugador superó a rival:
                superaciones[jugador] += 1
    retornar superaciones
```

Como se observa en el pseudocódigo, la complejidad temporal del algoritmo implementado por fuerza bruta es O(n^2) siendo n la cantidad de jugadores. Dicha complejidad se debe a la necesidad de iterar todos los jugadores, y sobre cada uno de ellos iterar nuevamente a todos para contar los superados. Una iteración simple es O(n) y en cada n se vuelve a iterar O(n). Por lo que se estaría haciendo O(n) n veces. O(n) x O(n) = O(n^2).

La complejidad espacial es O(n) ya que sólo se utiliza un diccionario como estructura para almacenar la cantidad de superaciones por jugador.

2. Proponer una solución utilizando la metodología de división y conquista que sea más eficiente que la propuesta anterior. (incluya pseudocódigo y explicación)

Pseudocódigo:

```
superaciones = {} diccionario para contar superaciones

resolucion_Div_Con(jugadores, superaciones):
    si hay más de un solo jugador:
        divido a los jugadores en dos sublistas (left y right)
        resolucion_Div_Con(left, superaciones) -> llamado recursivo
        resolucion_Div_Con(right, superaciones) -> llamado recursivo
        incremento = 0
        se hace un merge con left y right (ordenando por ranking anterior) -> un while
        si jugador_left.ranking_anterior < jugador_right.ranking_anterior:
            superaciones[jugador_left] += incremento
        si jugador_left_ranking_anterior > jugador_right_ranking_anterior:
            incremento += 1
            añadir a jugador_right a superaciones si no estaba (con 0 superaciones)
        si me quedan jugadores de la izquierda luego del while itero:
            superaciones[jugador_left] += incremento
        si me quedan jugadores de la derecha luego del while itero:
            añadir a jugador_right a superaciones si no estaba (con 0 superaciones)
```

En el diccionario superaciones queda la cantidad de rivales que superó cada jugador.

La solución propuesta por división y conquista se basa en el algoritmo MergeSort utilizado para ordenar arreglos. Se ingresa una lista con los jugadores ordenados por el ranking actual, y por MergeSort se realiza el ordenamiento de los jugadores según su posición en el ranking del año pasado. En el momento de hacer Merge (unir la mitad izquierda y derecha de forma ordenada) es donde es posible identificar la cantidad de jugadores que se superaron.

Durante el Merge, cuando se selecciona un jugador de la mitad derecha por sobre uno de la mitad izquierda, quiere decir que todos los jugadores de la izquierda tuvieron que

superar al de la derecha para estar ubicados delante de él. De esta forma se contabilizan las superaciones en cada Merge. Por lo tanto, simplemente debe ordenarse la lista de jugadores según su ranking del año anterior, y contabilizar las superaciones durante el proceso de Merge.

Mediante un conjunto de variables y un diccionario, es posible registrar las superaciones durante el Merge, sin tener que recorrer los jugadores más de las veces de las necesarias en un MergeSort ordinario.

Por cada Merge que se efectúa, se inicializa la variable "incremento" en cero. Por cada vez que se selecciona un jugador de la derecha (por ser menor que el de la izquierda), al incremento se le suma 1. "Incremento" corresponde a la cantidad de jugadores que superan los de la mitad izquierda durante el Merge. Luego, cuando se selecciona un jugador de la izquierda, se registra este incremento en el diccionario.

Al finalizar el ordenamiento, y luego de haber realizado todos los Merge necesarios, se obtiene un diccionario, con el registro total de superaciones.

La complejidad temporal del algoritmo es O(n*log(n)), siendo n la cantidad de jugadores. La misma es mucho mejor que la de fuerza bruta O(n^2). Al ser un mergeSort ordinario, en el cual se toma registro de superaciones, debería verificarse que el registro no altere la complejidad del Merge. Desarrollado en el ítem 3.

3. Realizar el análisis de complejidad temporal mediante el uso del teorema maestro.

Al plantear la resolución del problema con división y conquista, es posible dar la ecuación de recurrencia correspondiente:

a * T(n/b) + f(n)

a = cantidad de llamados recursivos.

b = proporción del tamaño original con el que se llama recursivamente.

f(n) = costo de partir y juntar.

En el algoritmo propuesto, a = 2 y b = 2. Por otro lado, el costo de partir y juntar, es el costo del Merge, más el costo de tomar registro de las superaciones efectuadas. Debido al uso de diccionarios, la complejidad de la inserción de un nuevo elemento es de O(1). Por lo tanto, no afecta el costo original de Merge O(n). Merge es O(n) porque recorre una sola vez, los n elementos a unir.

Para verificar qué caso de los planteados por el Teorema Maestro es el correspondiente al algoritmo planteado, debemos calcular: logba = log22 = 1

Según el Teorema Maestro:

Si f(n) = O(n ^ logba-e), e >0 -------> T(n) = O(n ^ logba)

Si f(n) = O(n ^ logba) -------> T(n) = O(n ^ logba * logn)

Si f(n) = O(n ^ logba+e), e >0 -------> T(n) = O(f(n))

En este caso, se cumple la 2da relación: f(n) = O(n) = O(n ^ log22) = O(n ^ 1) = O(n)

Con lo que se concluye, que el algoritmo propuesto para resolver el problema, tiene una complejidad temporal O(n * log n) siendo n la cantidad de jugadores.

4. Realizar el análisis de complejidad temporal desenrollando la recurrencia

El desenrollamiento de la recurrencia es similar al aplicado a MergeSort.

Se puede analizar cada "nivel" de la recurrencia hasta donde se encuentre el caso base. En el nivel 0 se tiene un problema de n elementos donde el costo de unir y separar es n. En el nivel 1 se tienen dos problemas de n elementos donde el costo de unir y separar es n/2 para cada uno. Por lo tanto, el costo es 2 * n/2 = n. En el nivel 2 se tienen 4 problemas con n/4 elementos donde unir y separar cada uno cuesta n/4. Lo cual sería 4 * n/4 = n. Esta lógica puede pensarse para infinitos niveles, en todos ellos se verá que el costo es n, lineal.

De esta manera el costo total del algoritmo es, n * cantidad de niveles.

Suponiendo una cantidad j de niveles, en el nivel j se tendrán 2 ^ j problemas. Teniendo en cuenta que j es el último nivel, los subproblemas allí son de 2 elementos cada uno. Si al comienzo se tiene n elementos, quiere decir que en el nivel j hay un total de n/2 problemas. De aquí se iguala 2 ^ j = n/2. Despejando se obtiene j = log n. Finalmente, el costo total del algoritmo corresponde a: n * cantidad de niveles = n * j. Siendo j = log n, se llega a que el mismo es O(n log n). Coincidente con la calculada por Teorema Maestro.

Desenrollamiento matemático:

T(n) = 2 * T(n/2) + n

T(2) = d –-> constante

T(n/2) = 2 * T(n/4) + n/2

T(n/4) = 2 * T(n/8) + n/4

…

T(n) = 2 [2 (2 { T(n/16) + n/8 } + n/4) + n/2] + n

termina siendo una sumatoria: $T(n) = 2^k * 1 + \sum_{i=0}^{k} (2^i * n)/2^i$

k al ser la cantidad de niveles, es log n.

$T(n) = 2^{logn} + n * \sum_{i=0}^{logn} 2^i / 2^i$

$T(n) = 2^{logn}$ + n * logn ----------> O(n * logn)

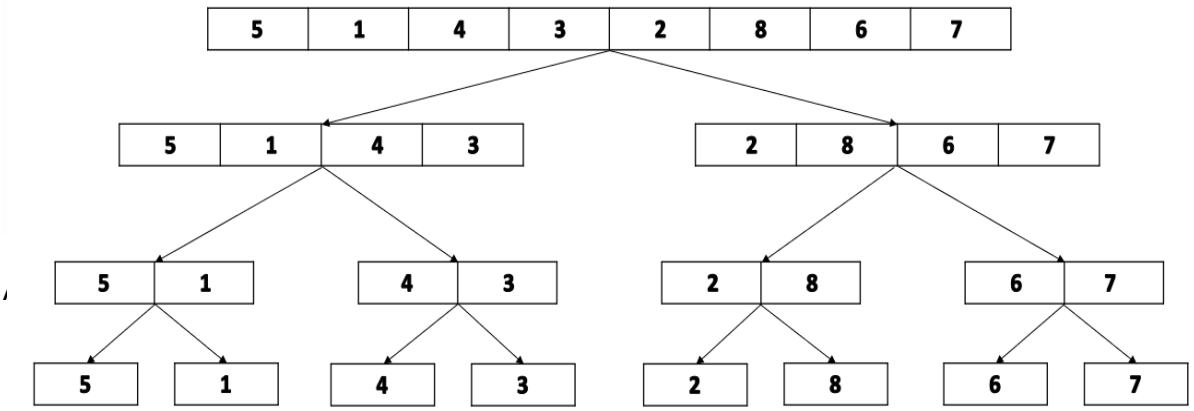
5. Analizar la complejidad espacial basándose en el pseudocódigo.

Observando el pseudocódigo, se utiliza como estructura un diccionario y se trabaja sobre la lista dada por parámetro. Para registrar las superaciones de cada jugador, se utiliza el diccionario en el que cada clave es un jugador y su valor la cantidad de rivales que superó. De esta forma, la complejidad espacial es O(n). Analizando la implementación, realizada en python, al no trabajar sobre la lista original con referencias a los índices, crear nuevas sublistas en los llamados recursivos, exige (en el peor de los casos) n log n de espacio nuevo. Siguiendo análisis de la complejidad temporal, sabemos que hay log n niveles, y en cada uno de ellos se utiliza n almacenamiento. Por lo tanto, el almacenamiento total destinado a las sublistas creadas durante la recursión es log n * n.

6. Dar un ejemplo completo del funcionamiento de su solución.

Ejemplo: `A,5|B,1|C,4|D,3|E,2|F,8|G,6|H,7`

Para simplificar la escritura, sólo se utilizarán los números para diagramar.



En el diagrama se representa el proceso de separación en mitades típico de MergeSort. Las sublistas izquierdas son las llamadas “left” y las sublistas derechas son las llamadas “right” (de cada llamado recursivo). Con la subdivisión de problemas diagramada, es posible hacer el seguimiento de cada Merge, de forma cómoda.

Se tiene un diccionario de superaciones vacío.

- Primer Merge, left = [5], right = [1], S = {} e Incremento = 0.

  - baja el 1, Incremento = 1, S = {“B”:0} se añade a S el 1 con 0 superaciones (es “B”).
  - baja el 5, S ={“B”:0, “A”:1} se añade a S el 5 con incremento superaciones (es “A”).

- Segundo Merge, left = [4], right = [3], S ={“B”:0, “A”:1} e Incremento = 0 (se resetea en cada Merge).

  - baja el 3, Incremento = 1, S ={“B”:0, “A”:1, “D”:0} se añade a S el 3 con 0 superaciones (es “D”).
  - baja el 4, S ={“B”:0, “A”:1, “D”:0, “C”: 1} se añade a S el 4 con incremento superaciones (es “C”).

- Tercer Merge, left = [1,5], right = [3,4], S ={“B”:0, “A”:1, “D”:0, “C”: 1} e incremento = 0.

  - baja el 1, (ya está en S e incremento es igual a 0)
  - baja el 3, incremento = 1 (ya está en S)
  - baja el 4, incremento = 2 (ya está en S)
  - baja el 5, S ={“B”:0, “A”:3, “D”:0, “C”: 1} se añade a S el incremento (es “A”).

- Cuarto Merge, left= [2] right= [8] S ={“B”:0, “A”:3, “D”:0, “C”: 1} incremento = 0

- baja el 2, S ={“B”:0, “A”:3, “D”:0, “C”: 1, “E”: 0} se añade a S el 2 con 0 superaciones (es “E”).
- baja el 8, S ={“B”:0, “A”:3, “D”:0, “C”: 1, “E”: 0, “F”: 0} se añade a S el 2 con 0 superaciones (es “F”).

- Quinto Merge, left= [6] right= [7] S ={“B”:0, “A”:3, “D”:0, “C”: 1, “E”: 0, “F”: 0} incremento = 0

- baja el 6, S ={“B”:0, “A”:3, “D”:0, “C”: 1, “E”: 0, “F”: 0, “G”: 0} se añade a S el 6 con 0 superaciones (es “G”).
- baja el 7, S ={“B”:0, “A”:3, “D”:0, “C”: 1, “E”: 0, “F”: 0, “G”: 0, ”H”:0} se añade a S el 7 con 0 superaciones (es “H”).

- Sexto Merge, left= [2,8] right= [6,7] S ={“B”:0, “A”:3, “D”:0, “C”: 1, “E”: 0, “F”: 0, “G”: 0, ”H”:0} incremento = 0

- baja el 2, (ya está en S)
- baja el 6, incremento = 1 (ya está en S)
- baja el 7, incremento = 2 (ya está en S)
- baja el 8, S ={“B”:0, “A”:3, “D”:0, “C”: 1, “E”: 0, “F”: 2, “G”: 0, ”H”:0} se añade a S el incremento (es “F”)

- Séptimo Merge, left = [1,3,4,5] right = [2,6,7,8] S ={“B”:0, “A”:3, “D”:0, “C”: 1, “E”: 0, “F”: 2, “G”: 0, ”H”:0} incremento = 0

- baja el 1, (ya está en S)
- baja el 2, aumento = 1 (ya está en S)
- baja el 3, S ={“B”:0, “A”:3, “D”:1, “C”: 1, “E”: 0, “F”: 2, “G”: 0, ”H”:0} se añade a S el incremento (es “D”)
- baja el 4, S ={“B”:0, “A”:3, “D”:1, “C”: 2, “E”: 0, “F”: 2, “G”: 0, ”H”:0} se añade a S el incremento (es “C”)

    - baja el 5, S ={“B”:0, “A”:4, “D”:1, “C”: 2, “E”: 0, “F”: 2, “G”: 0, ”H”:0} se añade a S el incremento (es “A”)
    - baja el 6 (ya está en S)
    - baja el 7 (ya está en S)
    - baja el 8 (ya está en S)
- S final = S ={“B”:0, “A”:4, “D”:1, “C”: 2, “E”: 0, “F”: 2, “G”: 0, ”H”:0}

  A,5|B,1|C,4|D,3|E,2|F,8|G,6|H,7

  “A” superó a 4 rivales.

  “B” superó a 0 rivales.

  “C” superó a 2 rivales.

  “D” superó a 1 rival.

  “E” superó a 0 rivales.

  “F” superó a 2 rivales.

  “G” superó a 0 rivales.

  “H” superó a 0 rivales.

Implementación

La implementación de ambas estrategias descritas previamente, estarán adjuntas o bien puede descargarse desde aquí.

Para ejecutar el programa, será necesario crear un archivo con extensión *.txt* con el siguiente formato:

$player_1_id,$previous_position|$player_2_id,$previous_position|$player_3_id,$previous_position…

El archivo contendrá un registro de los resultados de todos los jugadores. La información de cada jugador estará delimitada por el caracter “|”. Asu vez, la posición de la información del jugador en el registro, determinará su posición actual -empezando desde la izquierda hacia derecha-. Por último, la informacíón de cada participante estará formada por su identificador y la posición del año anterior, y estará delimitada en el archivo por el caracter “,”.

Una vez definido el archivo de registro de resultado de jugadores se debe ejecutar el siguiente comando por consola:

python main.py mi_archivo.txt

por default se ejecutará la solución mediante el algoritmo de división y conquista. En caso de querer usar la estrategia de fuerza bruta, ejecutar el siguiente comando:

python main.py mi_archivo.txt by_brute_force

Una vez implementadas ambas estrategias, y testeadas con distintas instancias del problema (aquí están los test de cada implementación) se realizaron pruebas de performance. Para ello se crearon colecciones aleatorias de n elementos y se procedió a graficar el tiempo necesario para procesar cada solución (aquí están los test de performance).

En cada test de performance, se define un tamaño máximo de elementos de la colección y un tamaño mínimo, correspondiente a 100 elementos. Luego, se irá incrementando el tamaño del arreglo de a 500 elementos hasta llegar al máximo, generando en cada iteración un nuevo arreglo aleatorio. En las figura 1 se puede evidenciar el tiempo de procesamiento necesario para cada iteración, para un máximo de 40000 elementos.

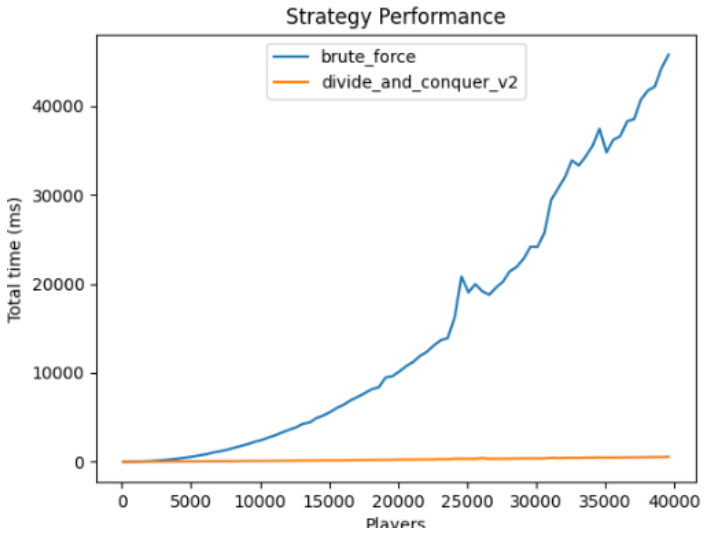
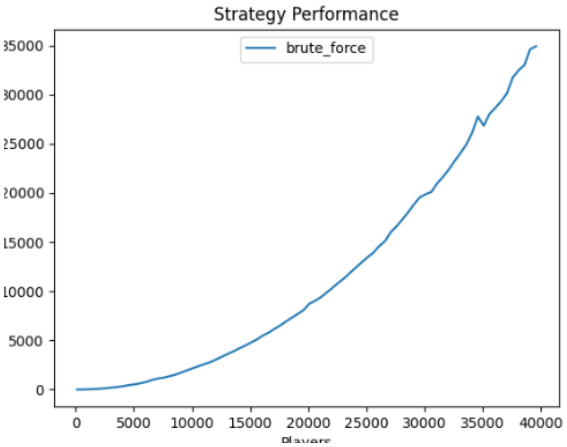
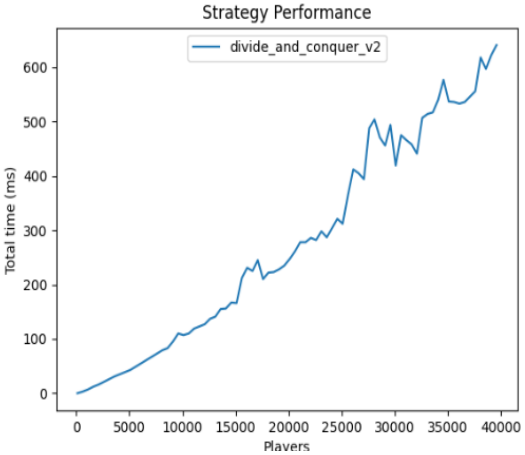


Figura 1: comparativa entre fuerza bruta y división y conquista para 40000 elementos





Los resultados mostrados en la figura 1, validan el orden teórico de cada estrategia, dado qué el tiempo requerido por la solución de fuerza bruta es similar a la mitad de una parábola correspondiente a una función cuadrática. Por el contrario, la curva correspondiente al tiempo necesario para división y conquista parece “no crecer en el infinito”, es decir qué su pendiente es muy chica, correspondiendose con una curva de tipo logarítmica.

Parte 2

Hicimos 9 tests, cada uno con un input asociado y sus respectivas imágenes en el path test/images.
Hay 3 scripts en el root del proyecto

- run-example.sh : Ejecuta el código para un caso en particular.
- run-example-v.sh: Ejecuta en modo verboso imprimiendo el paso a paso de la construcción de la matriz y de la estructura de nodos candidatos a ser parte de un ciclo negativo, para los 9 ejemplos.
- run-test.sh: ejecuta todos los test.

Ejecución:
En el directorio de los fuentes de la solución ejecutar:

Para modo normal:

```
_> python3 ./main.py -f “path_to_file”
```

Para modo verbose:

```
_> python3 ./main.py -v True -f “path_to_file”
```

1. Pseudocódigo

```
n = cantidad de nodos
Grafo = input de nodos con sus predecesores y costos

Optimo = matriz
Desde l=0 a n      //O(n)
  Optimo[l][0] = 0

Desde v=1 a n-1    //O(n); v=0 es nodo inicial
  Optimo[0][v] = ∞
```

```
Desde l=1 a n    //O(n)
  Desde v=1 a n-1  //O(n)
    Optimo[l][v] = Optimo[l-1][v]
    Por cada p de v    //O(m); p es predecesor
       costo = Optimo[l-1][p] + w(p,v)
       Si costo < Optimo[l][v]
         Optimo[l][v] = costo

candidatos = []
Desde v=0 a n-1    //O(n)
  Si Optimo[n][v] <  Optimo[n-1][v]
    agregar nodo a candidatos

Si candidatos no es vacio
  ciclo = []
  pesos = []
  Por cada c en candidatos   //O(n) a lo sumo
    Para v del grafo    //O(n)
      Si existe una arista de v que llega a c
        agregar a v a ciclo
        agregar w(c,v) a pesos
        si v ya fue agregada a ciclo
          retornar ciclo, suma(pesos)
```

Estructuras de datos utilizadas

- Grafo es un diccionario que contiene como clave aquellos nodos a los cuales les llega una arista, y como valor un array de todos sus predecesores con el costo de la arista que los une.
- Nodes es un vector de todos los nodos del grafo.
- Candidatos a ciclos negativos es un vector que contiene los nodos que pertenecen algún ciclo negativo

Explicación del algoritmo

Nos basamos en el algoritmo de Bellman-Ford visto en el módulo de programación dinámica de la materia, para identificar los caminos de costo negativo de un grafo, construyendo la matriz de n(nodos) columnas por 0 a n filas.

Una vez construida, sabemos que los valores que mejoraron el costo en la última fila, son aquellos nodos candidatos a pertenecer a un ciclo negativo, y si hay mejora garantiza que existe un ciclo negativo.

Queda construir una nueva estructura que incluya el costo de todas las aristas que forman parte del ciclo. Para ello, es necesario desandar el camino de cómo se llegó a dicho vértice, e ir sumando los costos de las aristas que forman parte de dicho ciclo negativo.

2. Complejidades

Temporal

No tuvimos en cuenta el costo de acceder a los elementos de la matriz, ya que es de O(1).

Así como tampoco de imprimir los valores en modo verbose.

El primer y segundo término corresponde a inicializar la matriz, luego el tercero armar la matriz de caminos mínimos, según el algoritmo de Bellman-Ford. El cuarto término corresponde a armar la estructura que guarda los costos de los ciclos negativos, donde se itera el vector de nodos.

$$O(n) + O(n) + O(n * n * m) + O(n) + O(n * n) = O(n * n * m)$$

Espacial

No consideramos la complejidad del diccionario en el cual cargamos la información del input.

El primer término corresponde al tamaño de la matriz de caminos mínimos. El segundo corresponde al tamaño del vector de nodos del grafo. El último término corresponde al tamaño del array que guarda los vértices que son parte de ciclos negativos, es a lo sumo m aristas.

$$O(n * n) + O(n) + O(n) + O(m) = O(n * n)$$

Comparación de complejidad teórica y la implementación

Tomamos como solución teórica el pseudocódigo.

Las complejidades espaciales difieren levemente en un término de O(n), siendo la implementación un poco más cara que la solución teórica, por el hecho de tener que sumar los pesos, en lugar de sumar los costos a medida que recorremos la estructura de ciclos, lo hacemos al final.

	Solucion Teorica	Implementación
Complejidad temporal	$O(n * n * m)$	$O(n * n)$
Complejidad espacial	$O(n * n)$	$O(n * n)$

Parte 3

1. Describa brevemente en qué consiste cada una de ellas. ¿Cuál es la mejor de las 3? ¿Podría elegir una técnica sobre las otras?

Estrategia Greedy: Es un paradigma algorítmico orientado a problemas de optimización donde se busca maximizar o minimizar alguna cantidad. Divide el problema en subproblemas que presentan jerarquía entre ellos. Los subproblemas se resuelven iterativamente mediante una elección heurística y dan lugar a nuevos subproblemas de mayor jerarquía. La estrategia greedy resolverá los subproblemas iterativamente, lo que llevará a la solución óptima global. No todos los problemas pueden solucionarse mediante una estrategia de tipo greedy, se requiere que el problema a resolver presente 2 propiedades: *elección greedy* y *subestructura óptima*. Si no hay una elección greedy y una subestructura óptima no se podrá implementar una está estrategia.

La elección greedy requiere tomar de manera arbitraria para cada subproblema una elección óptima local, qué contribuya a llegar a la elección óptima global. La elección local se realiza con los datos que se tienen en ese punto, no se consideran los subproblemas

anteriores ni posteriores. La subestructura óptima se encuentra presente cuando la solución óptima global del problema contiene las soluciones óptimas de los subproblemas.

La complejidad de las estrategias greedy de este tipo no radica solo en determinar la subestructura óptima y la elección greedy que resuelva el problema, sino en demostrar que dicha estrategia greedy lleva efectivamente a una solución óptima para toda instancia del problema.

Programación dinámica: Al igual que la estrategia anterior, es una metodología de resolución de problemas orientados a optimización. Divide el problema en subproblemas, con ciertas similitudes a greedy ya que tienen jerarquías entre ellos, pero se diferencia al utilizar cada subproblema en subproblemas mayores. Para poder resolver un problema mediante programación dinámica este debe presentar dos propiedades: *subestructura óptima y subproblemas superpuestos*.

Al igual que en greedy, la subestructura óptima está presente si la solución óptima global contiene las soluciones de sus subproblemas. Los subproblemas superpuestos están en un problema si en la resolución de los subproblemas aparecen subproblemas de menor jerarquía previamente calculados.

Las estrategias de programación dinámica se pueden resolver de forma recursiva, por lo que se pueden representar mediante una ecuación de recurrencia. Existen uno o varios términos base desde los que se parte para calcular los siguientes.

Una característica fundamental a la hora de aplicar programación dinámica es la *memorización*, la cual consiste en almacenar los resultados de los subproblemas calculados. Esto permite evitar calcular varias veces un mismo subproblema que aparece de manera repetida (subproblemas superpuestos). Así se reduce de manera significativa la cantidad de subproblemas a calcular reduciendo en consecuencia la complejidad temporal de la solución. A la hora de implementar un algoritmo que siga esta estrategia es conveniente utilizar un algoritmo iterativo, dado que facilita la memorización.

División y conquista: Es otra metodología de resolución de problemas, esta divide un problema en subproblemas que a diferencia de greedy y programación dinámica no tienen jerarquía entre ellos, sino que son de igual naturaleza y menor tamaño. Cada subproblema es resuelto de manera recursiva hasta un caso base, con todos los subproblemas resueltos se combinan los resultados en una solución general. No todo problema se puede resolver mediante un algoritmo de división y conquista, al dividir el problema en subproblemas estos deben ser de la misma naturaleza del problema original y por lo tanto deben poder dividirse de manera recursiva en subproblemas de menor tamaño también de la misma naturaleza que

el problema original, este proceso continúa recursivamente hasta tener problemas suficientemente simples para ser resueltos de manera directa. Cada subproblema debe poder resolverse por separado sin necesidad de la totalidad de los elementos. Con los subproblemas resueltos estos se combinan dando como resultado la solución del problema original.

En general se puede aplicar a ciertos problemas donde la solución por fuerza bruta ya tiene una solución polinómica pero se busca mejorar la eficiencia. Como los problemas se resuelven recursivamente es necesario resolver una relación de recurrencia para analizar la complejidad del algoritmo, además este tipo de estrategias permiten aprovechar el procesamiento en paralelo siempre y cuando cada problema/subproblema se divida en dos o más subproblemas a resolver.

Greedy vs. D&C vs. programación dinámica

En principio la elección de una técnica sobre las otras dependerá de la naturaleza del problema a resolver. Como ya se mencionó cada estrategia requiere que el problema cumpla con ciertas propiedades para ser resuelto, un problema que no permite dividir el problema en subproblemas cuya solución lleve a la solución global no podrá ser resuelto mediante una estrategia greedy ni mediante programación dinámica; mientras que un problema que no se pueda dividir en subproblemas independientes de la misma naturaleza que el problema original no podrá ser resuelto mediante un algoritmo de D&C.

En el caso que múltiples estrategias puedan ser aplicadas a un mismo problema habrá que evaluar las ventajas y desventajas de cada una, ya sea la complejidad temporal o espacial de los algoritmos que implementan cada estrategia, o evaluar el riesgo de un error del tipo stack overflow al que puede llevar un algoritmo de tipo D&C frente al beneficio de implementar una solución que aproveche el procesamiento en paralelo.

En algunos casos una solución mediante D&C presenta subproblemas que se repiten (subproblemas superpuestos) en estas situaciones se puede optar por utilizar una técnica de memorización como la que se utiliza en programación dinámica para mejorar la complejidad temporal a cambio de un posible aumento en la complejidad espacial. Se podría incluso modificar el algoritmo para que tenga un enfoque bottom-top y sea iterativo, convirtiéndolo así en uno de tipo de programación dinámica. Un problema que se puede resolver mediante división y conquista, pero mejora su eficiencia al aplicar un algoritmo de programación dinámica es por ejemplo el cálculo del enésimo número de Fibonacci.

En algunos problemas donde greedy no sea aplicable para hallar la solución óptima y una estrategia de programación dinámica si lo sea se podría aún optar por implementar una

estrategia greedy, que ofrezca una solución no óptima, pero se acerque a la misma con un bajo coste frente a una estrategia de programación dinámica que sea demasiado costosa. Un ejemplo de esta situación es el problema del viajante, donde una estrategia greedy ofrece una solución no óptima en tiempo polinomial, mientras que una estrategia de programación dinámica ofrece una solución óptima, pero con complejidad temporal exponencial.

2. Un determinado problema puede ser resuelto tanto por un algoritmo Greedy, como por un algoritmo de Programación Dinámica. El algoritmo greedy realiza N^3 operaciones sobre una matriz, mientras que el algoritmo de Programación Dinámica realiza N^2 operaciones en total, pero divide el problema en N^2 subproblemas a su vez, los cuales debe ir almacenando mientras se ejecuta el algoritmo. Teniendo en cuenta los recursos computacionales involucrados (CPU, memoria, disco) ¿Qué algoritmo elegiría para resolver el problema y por qué?

Para determinar la elección del algoritmo se debería tener en cuenta no solo los recursos computacionales, sino también el tamaño de la entrada. Supongamos una entrada N del orden 10^3, si el CPU pudiese procesar 10^8 operaciones por segundo al algoritmo greedy le tomaría unos 10 segundos solucionar el problema, mientras que el algoritmo de programación dinámica podría resolverlo en centésimas de segundo usando (al menos) 1 Mb de memoria (si por ejemplo almacenase 1 byte por cada subproblema), en este caso dado el menor tiempo y el bajo uso de memoria por parte del algoritmo de programación dinámica este sería una mejor elección.

Supongamos ahora una entrada del orden de 10^4 . El algoritmos greedy tardaría 3 horas en resolver el problema, mientras que al algoritmo de programación dinámica le llevaría virtualmente un segundo, y si por cada subproblema almacena un byte entonces necesitaría 100 Mb de memoria aproximadamente; pero ¿qué sucede si almacena más de 1 byte por subproblema? Dado que se están trabajando con matrices podemos suponer que el valor de la entrada N proviene del tamaño de la matriz, por ejemplo, podría tratarse de una matriz de enteros cuadrada de NxN, y siendo las matrices estructuras en general costosas para ser copiadas o almacenadas se puede realizar mayor análisis que involucre el costo espacial. Si por cada subproblema debieran almacenarse submatrices de 2x2 (valor arbitrario para el cálculo), la cantidad de memoria necesaria pasaría de 100 Mb a 1.6 Gb. En este caso la

memoria RAM de una computadora moderna no tendría problemas para almacenar los subproblemas que requiere el algoritmo por lo que la elección sigue siendo el algoritmo de programación dinámica.

Manteniendo una entrada del orden de 10^4 requeriría que por cada subproblema se almacenen 100 bytes para que el segundo algoritmo requiera una memoria total mayor a 10 Gb, llegado este caso un sistema con menos de 8 Gb de memoria RAM ya no podría almacenar las soluciones de los subproblemas y se empezaría a hacer uso del disco, cuya velocidad es considerablemente inferior a la de la memoria RAM y requeriría que el CPU se dedique a copiar información al disco, incrementando el tiempo que le llevaría al algoritmo resolver el problema, pero tal vez aun inferior a las 3 horas que le lleva al algoritmo greedy resolver el problema. Si la entrada siguiese incrementando en tamaño o fuese mayor la cantidad de bytes almacenados por subproblema del algoritmo de programación dinámica, llegaría un punto donde la cantidad de memoria requerida alcanzaría el orden de los terabytes, resultando en un excesivo costo en memoria y haciendo el algoritmo de programación dinámica aún más lento que el algoritmo greedy debido a las operaciones de escritura en disco.

En el siguiente cuadro se puede observar una comparativa entre el tiempo que le tomaría a cada estrategia resolver el problema según el orden de la entrada. También se puede ver la memoria que necesitaría el algoritmo de programación dinámica según el orden de la entrada y la memoria necesaria por subproblema.

Entrada	Tiempo (10^8 operaciones por segundo)		Memoria estrategia prog. dinámica	
	Greedy	Prog. Dinámica	1 byte por subproblema	100 bytes por subproblema
10^1	0,01 ms	0,001 ms	100 bytes	10 kb
10^2	0,01 s	0,1 ms	10 kb	1 Mb
10^3	10 s	0,01 s	1 Mb	100 Mb
10^4	3 h	1 s	100 Mb	10 Gb
10^5	116 días	100 s	10 Gb	1 Tb
10^6	317 años	3 h	1 Tb	100 Tb

En conclusión, la elección de un algoritmo u otro dependerá del tamaño de la entrada y de la memoria almacenada por el algoritmo de programación dinámica en cada subproblema.

Referencias

1- https://github.com/jorgejcabrera/tda-divide-and-conquer/

2. https://github.com/diego-wl/tda-dynamic-programming