Teoría de Algoritmos

Trabajo Práctico 1

Año: 2024 1° Cuatrimestre

Grupo: Capybaras Salvajes

Integrante	Padrón
Sebastián Outeiro	92108
Santiago Marinaro	97969
Agustin Cavo	95971
Tomas Baqueiro	111146
Tomas Gonzalez	109717

Año: 2024 1° Cuatrimestre

Entrega: 1

Fecha: 15/04/2024

Índice

Parte 1: La campaña

1.1°) Solución Propuesta

Obtener el listado de influencers con mayor penetración de mercado corresponde a un problema de optimización cuyo espacio de soluciones son todas las combinaciones posibles de influencers.

Debido a que cada influencer posee un listado de otros influencers con los que no trabajaría, tomaremos esta información como la propiedad de corte. Por lo tanto, nuestra **función límite** deberá evaluar al agregar un influencer al listado de solución, si el mismo fue descartado por otro influencer que ya pertenece al conjunto solución. En caso afirmativo, no podremos agregar dicho influencer al listado.

Por otro lado, dado que los influencers tienen distintos valores de penetración de mercado, realizaremos un modelo de árbol donde en cada nivel deberemos determinar si incluimos o no a un influencer dentro del listado solución. Dicho árbol, tendrá como raíz el listado vacío sin ningún influencer, y estará ordenado de forma tal que a medida que descendamos en el árbol encontraremos influencers con menor valor de penetración de mercado.

Para esto, utilizaremos una **función de costo** que nos dará un estimado de la máxima penetración de mercado que podrá tener una rama del árbol, obteniendo la penetración de mercado acumulada sumado al valor de penetración de dicho nodo, y sumando la cantidad de influencers remanentes por ser agregados por el valor de penetración del próximo influencer. En los escenarios donde el valor obtenido por la función de costo sea menor a la mejor solución ya encontrada, se podrá realizar la poda de dicha rama. Además, la función de costo nos servirá para elegir qué rama explorar primero.

Obtendremos entonces los siguientes valores y funciones:

- Listado mejor Solución Encontrado: **S**. Su estado inicial será: S = []
- Valor de penetración acumulada: **k**
- Cantidad de influencers totales: **n**
- Listado de influencers descartados: D. Su valor inicial será D = []
- Influencers remanentes: i_r
- Valor de penetración de mercado del influencer: v_1
- Valor de penetración de mercado del influencer siguiente: $v_{(i+1)}$
- **Función de Costo**: $l = k + i_r * v_{(i+1)}$

1.2°) Pseudocódigo

```
solución:
        influencers = listado vacío
        penetracionAcumulada = 0
        influencersDescartados = set vacío

campaña(influencers):
        influencersOrdenados = ordenar(influencers)
        solución = nuevo objeto solución
        backtrack(solución, influencersOrdenados, 0 )

backtrack(solución, influencers, posición):
        si la posición es igual a la cantidad de influencers
                devuelvo la solución

        mejorSolución = solución
        influencerActual = influencers en posición

        si solución puedo agregar influencerActual:
                soluciónNueva = solución + influencerActual
                costoNuevo = costo(soluciónNueva, influencers, posición)
                si costoNuevo > solución.penetraciónAcumulada:
                        mejorSolución = backtrack(soluciónNueva, influencers, posición + 1)


        costoActual = costo(solución, influencers, posición)
        si costoActual > solución.penetracionAcumulada:
                mejorSolución = backtrack(solución, influencers, posición + 1)

        devuelvo mejorSolución
```

1.3°) Complejidad

Temporal

La complejidad temporal de este algoritmo estará determinada por el peor escenario, donde sea necesario explorar todos las soluciones posibles. Al ser un árbol binario de profundidad n, tendremos un total de 2^n estados posibles.

Otras operaciones como el ordenamiento de los influencers tendrá una complejidad temporal de O(nlogn).

La complejidad final por lo tanto será de $O(2^n)$ en el peor de los casos.

Espacial

Debido al algoritmo recursivo, en el peor de los escenarios, tendremos 2^n estados posibles y por cada uno de ellos tendremos como mucho dos objetos solución que poseen en el peor de los casos listados internos con todos los influencers. Por lo tanto tendremos un $O(2 * 2^n * 2n) = O(n*2^{(n + 2)}) = O(n*2^n)$

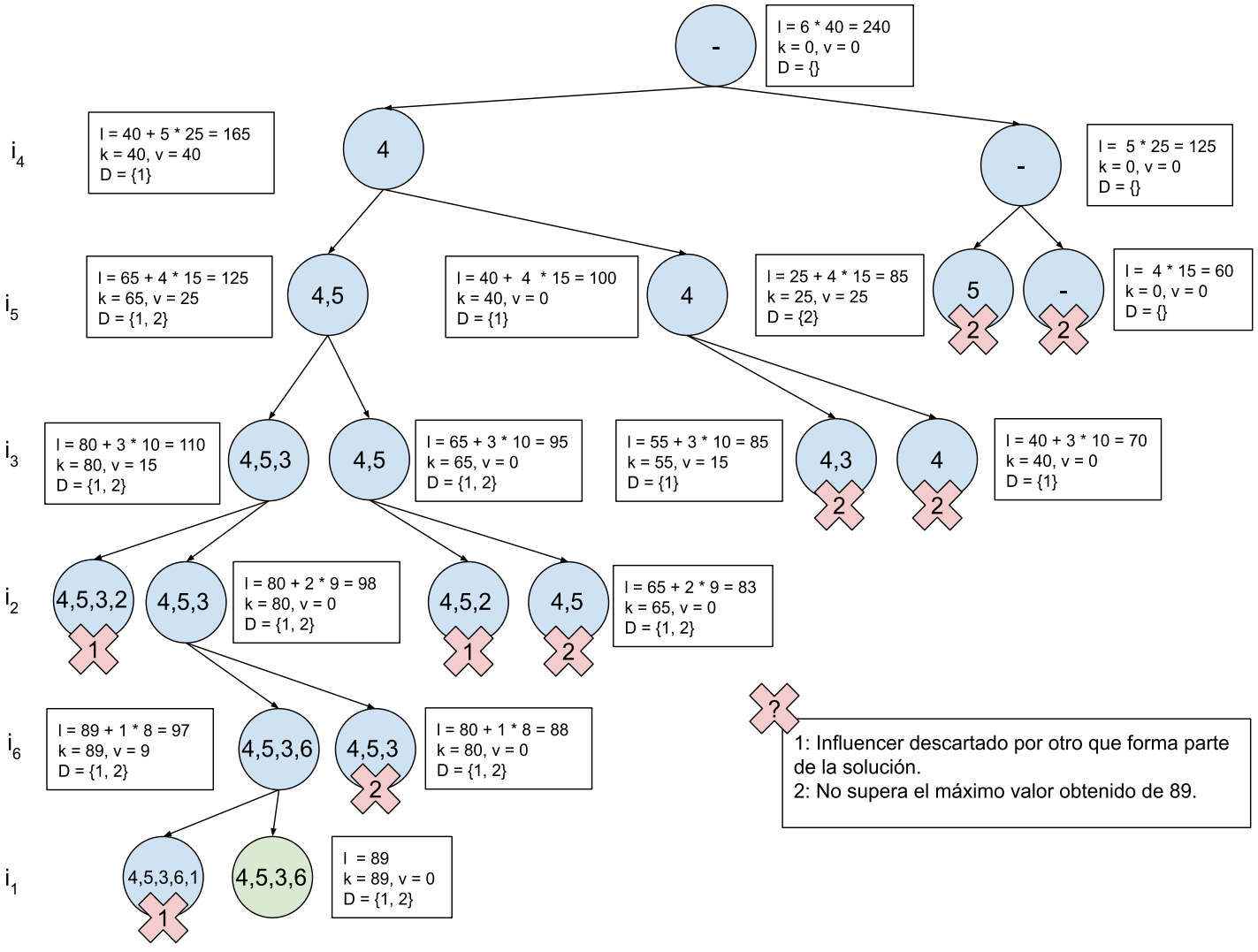
1.4°) Ejemplo

Tomaremos como ejemplo el siguiente caso, donde tenemos, los siguientes valores: (i, nombre, valor de penetración de mercado, listado de influencers con los que no trabajaría):

1,Cacho,8,3,4 2,Lucho,10,5 3,Suncho,15,1 4,Pucho,40,1 5,Tucho,25,2 6,Fercho,9

Ordenando los influencers por penetración de mercado, encontraremos el siguiente orden:
{4, 5, 3, 2, 6, 1}

A continuación desarrollamos el modelo de árbol donde como se describió en la solución, en cada nivel evaluaremos a un influencer en el orden decreciente en cuanto a su penetración de mercado, y evaluaremos en cada nivel la posibilidad de agregar o no agregar a dicho influencer. Luego, para cada nodo realizaremos los cálculos de las funciones de límite y de costo, y evaluaremos si debemos realizar alguna poda o si debemos continuar explorando sus descendientes.



Por lo que podemos ver, la solución encontrada es S = {4, 5, 3, 6} donde el total de penetración de mercado lograda será de 89 puntos. Teniendo entonces a los influencers, S = { Pucho, Tucho, Suncho, Fercho }.

1.5°) Código

El código se encuentra en el siguiente repositorio en la carpeta tp1/ej1 y se adjunta el .zip con el código fuente.
https://github.com/sebastian-outeiro/tda

Ejecución - JAR

Se adjunta en los archivos un JAR pre-compilado para su rápida ejecución. En caso de que su ejecución no funcione, se debera proceder a compilar el mismo nuevamente.

1°) Ir a la carpeta:

/tp1/ej1

2°) Abrir una consola en la ubicación y ejecutar el comando

java -jar ej1.jar <archivo>

3°) Ejemplos

java -jar ej1.jar ejemplo.txt

Compilación - JAR

En caso de que sea requerido re-compilar el JAR se pueden seguir los siguientes pasos:

1°) Ir a la carpeta:

/tp1/ej1

2°) Compilar con maven:

mvn compile

3°) Armar el package con maven

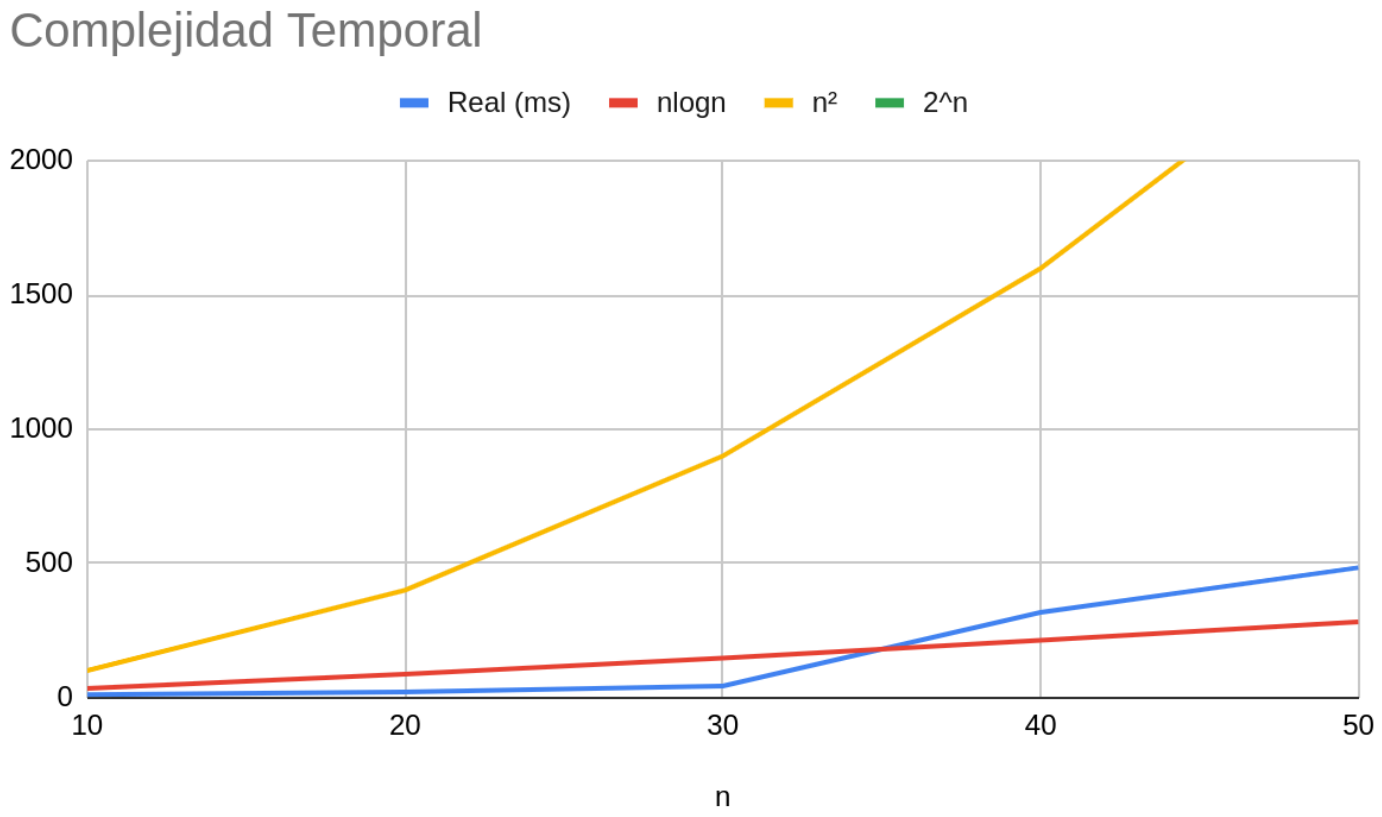
mvn package

4°) El .jar ejecutable se creara en la carpeta /target. Se podria ejecutar el mismo con el siguiente comando:

java -jar target/*.jar ejemplo.txt

1.6°) Demostración Complejidad

N (Elementos)	Real (ms)	nlogn	n^2	2^n
10	10	33.22	100	1024
20	20	86.44	400	1048576
30	43	147.21	900	1.07E+09
40	317	212.88	1600	1.10E+12
50	484	282.19	2500	1.13E+15



Podemos observar que la complejidad real se encuentra muy cercana a la complejidad de $O(n\log_2(n))$, la cual es del orden de complejidad del algoritmo de ordenamiento utilizado.

A su vez, se puede observar que el algoritmo se comportará distinto dependiendo de la cantidad de influencers descartados promedio por cada influencer, mejorando la velocidad de procesamiento en los casos donde el promedio de influencers descartados sea más alto.

Podríamos pensar un escenario, donde todos los influencers descarten al resto de los influencers. De esta manera, sólo podremos elegir un influencer y la mejor combinación se daría de encontrar el que tenga la mayor penetración de mercado, por lo que ordenando los mismos y tomando el máximo lograríamos resolver el problema en $O(n\log_2(n))$. Podemos considerar a esto como el mejor caso posible en complejidad temporal.

Parte 2: Reunión de camaradería

2.1°) Solución Propuesta:

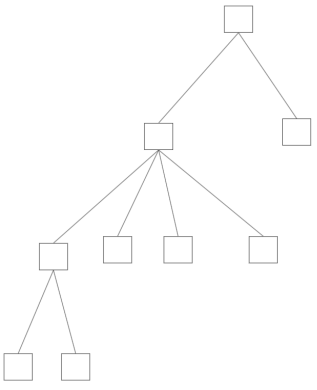
Para resolver este problema lo que pensamos es ir recorriendo el árbol con el método postorden y vamos invitando a los que no tengan hijos o no tengan hijos invitados.

2.2°) Pseudocódigo

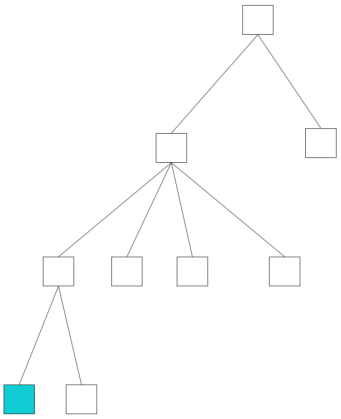
```
arbol = inicializar árbol con los datos del organigrama

recorrer_postorden(nodo):

 si nodo tiene hijos:

      for hijo in nodo.hijos:

          recorrer_postorden(hijo)

      si los hijos no están invitados:

           invitar nodo

   else:

           invitar nodo

recorrer_postorden(arbol.raiz)
```

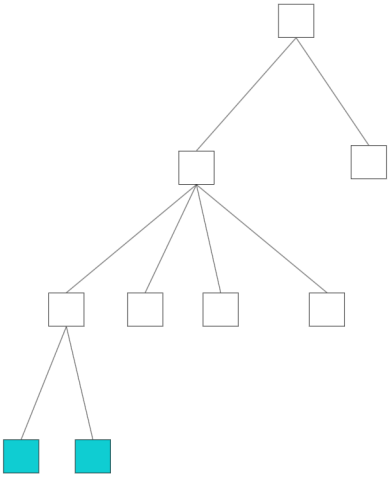
2.3°) Ejemplo

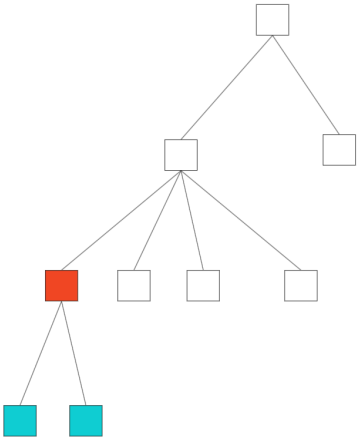
- Blanco = no visitado
- Azul = Invitado
- Rojo = visitado pero no invitado

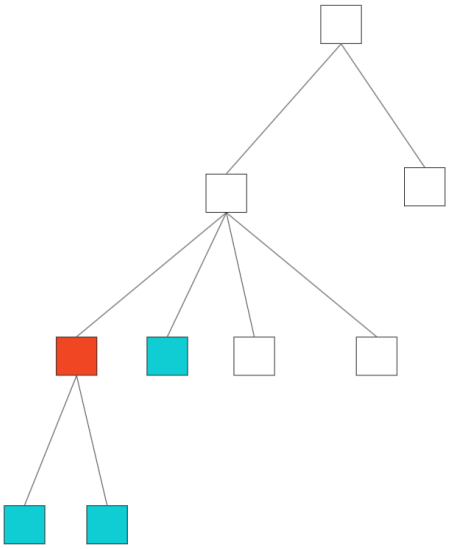


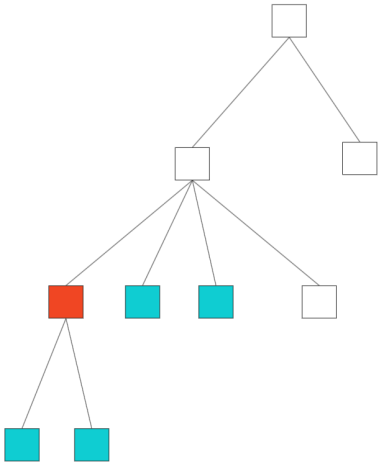
Estado Inicial: Todos sin invitar

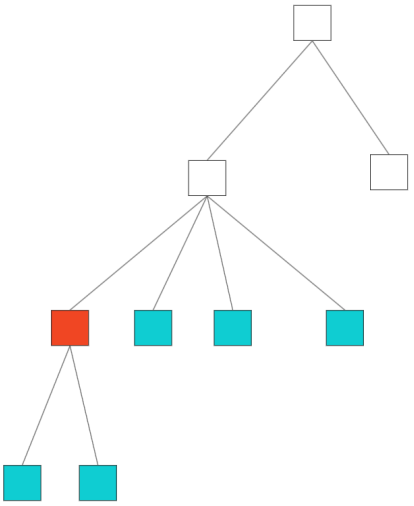


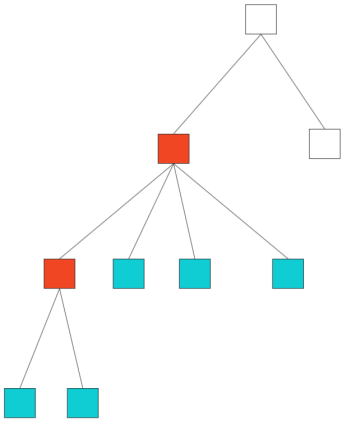


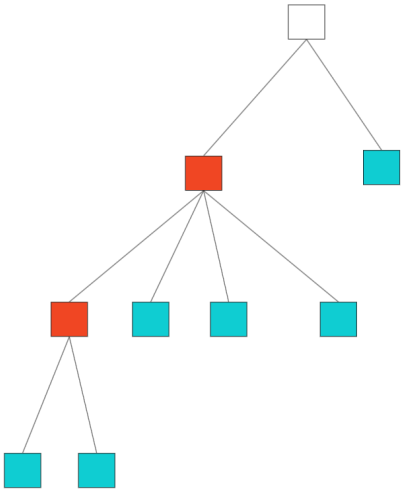


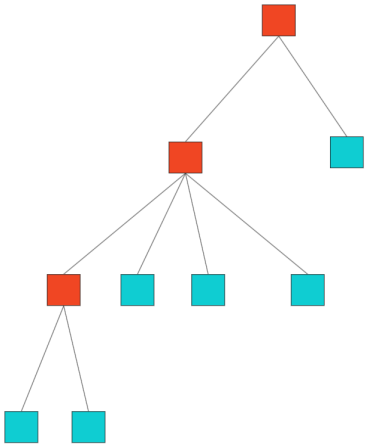












Complejidad temporal:

- recorrido postorden O(n)
- invitar O(1)
- chequear si al menos un hijo no está invitado en el peor de los casos O(n-1) = O(n)
- costo total: O(n)xO(n) = O(n)

Complejidad espacial:

- Almacenamiento de árbol: O(n)

2.4°) Justificación Greedy

La decisión greedy fue empezar a recorrer desde las hojas invitándolas y luego ir para arriba. Esta decisión es factible e irrevocable.

2.5°) Demostración

Cualquier nodo que no es una hoja siempre tiene uno o más hijos. Al empezar desde las hojas, nodos que no tienen hijos, nos aseguramos que invitamos a la misma cantidad o más de la que no podemos invitar.

Parte 3: La regionalización del campo

3.1°) Solución Propuesta:

1. Primero se divide en 4 sectores la matriz actual, haciendo que la matriz “campo” sea del tamaño de la anterior potencia de dos hasta llegar a la mínima permitida por enunciado y posible de resolver que es 2x2.

2. Cada llamada de la función recursiva recibe un espacio vacío que debe respetar para permitir la conjunción de las regiones adyacentes cuando la recursividad está terminando, el primer espacio vacío que se tiene que respetar es el silo.

3. Cada región comprueba si tiene el que debería quedar vacío de ser así lo pasa a la siguiente recursión sino se le pasa uno por default de acuerdo al sector que se encuentra de la matriz siempre basado en su posición.

4. Cuando termina la recursión de un sector basado en que espacio vacío fue recibido se rellenan los espacios vacíos de la matriz.

3.2°) Relación de recurrencia:

T(n)= A *T(n/B) + O(n^C)

A: cantidad de llamados recursivos
B: proporción del tamaño original con el que llamamos recursivamente
C: el costo de partir y juntar (todo lo que no es recursivo)

Cada vez que parto la matriz para reducir su tamaño y rellenar, llamó una función de recursividad para ese nuevo sector sea analizado => A=4.

El problema principal se divide en 4 subproblemas a resolver y así sucesivamente hasta el caso base => B=4.

El costo de llenar la matriz de 2x2 evitando un espacio en concreto es O(4) y los agujeros a rellenar siempre son de una L, la cual por la disposición previamente echa sabemos cual es O(3)=> O(4)+O(3) (por propiedades de notación O) => O(1)=> llenar toda la matriz es $O(n^2)$

T(n)= 4 *T(n/4) + O(n^0) => T(n)= 4 * T(n/4) + O(n^2)

3.3°) Pseudocódigo:

Main:

```
Set fila silo= parametro 1
set col silo = parametro 2
Set campo = matriz[parametro 0][parametro 0]

desde 0 a parametro 0, i++
        desde 0 a parametro 0,j++
```

```
                    Matriz[i][j] = “Vacio”

    Matriz[fila silo][col silo] = “Silo”

    division(0,parametro 0,0,parametro 0, silo,campo)

    mostrar campo
```

division(inico fila, fin fila, incio col, fin col, vacio):

```
    set mediofila=iniciofila + (finfila-iniciofila)/2
    set mediocol=iniciocol + (fincol-iniciocol)/2

    si (finfila-inciofila)==2:
        llenarmenor(iniciofila,inciocol,vacio)
        return

    si vacio esta_en_el_sector_superior_izq
        division(iniciofila,mediofila,iniciocol,mediocol,vacio)
        rellenar(caso 0,iniciofila,finfila,iniciocol,fincol)
    sino
        division(iniciofila,mediofila,iniciocol,mediocol,(mediocol-1,mediofila-1))

    si vacio esta_en_el_sector_superior_der
        division(iniciofila,mediofila,mediocol,fincol,vacio)
        rellenar(caso 1,iniciofila,finfila,iniciocol,fincol)
    sino
        division(iniciofila,mediofila,mediocol,fincol,(mediocol,mediofila-1))

    si vacio esta_en_el_sector_inferior_izq
        division(mediofilas,findinla,iniciocol,mediocol,vacio)
        rellenar(caso 2,iniciofila,finfila,iniciocol,fincol)
    sino
        division(mediofilas,findinla,iniciocol,mediocol,(mediocol-1,mediofila))

    si vacio esta_en_el_sector_inferior_der
        division(mediofilas,finfila,mediocolumnas,fincol,vacio)
        rellenar(caso 3,iniciofila,finfila,iniciocol,fincol)
    sino
        division(mediofilas,finfila,mediocolumnas,fincol,(mediocol,mediofila))

    return
```

llenarmenor (iniciofila,iniciocol,vacio):

```
    desde iniciofila a iniciofila+1, i++
        desde iniciocol a iniciocol+1,j++
            si es el vacio
                continue
            campo[i][j]= “region” + cantidadregiones

    aumentar la cantidad de regiones del campo en 1
```

```
        return

rellenar(caso,iniciofila,finfila,iniciocol,fincol):
        set mediofila=iniciofila + (finfila-iniciofila)/2
        set mediocol=iniciocol + (fincol-iniciocol)/2

        si es el caso 0:
                llenar campo[mediofila-1][mediocolumna]=  “region” + cantidadregiones
                llenar campo[mediofila][mediocolumna-1]=  “region” + cantidadregiones
                llenar campo[mediofila][mediocolumna]=  “region” + cantidadregiones
        si es el caso 1:
                llenar campo[mediofila][mediocolumna-1]=  “region” + cantidadregiones
                llenar campo[mediofila][mediocolumna]=  “region” + cantidadregiones
                llenar campo[mediofila-1][mediocolumna-1]=  “region” + cantidadregiones
        si es el caso 2:
                llenar campo[mediofila-1][mediocolumna]=  “region” + cantidadregiones
                llenar campo[mediofila][mediocolumna]=  “region” + cantidadregiones
                llenar campo[mediofila-1][mediocolumna-1]=  “region” + cantidadregiones
        si es el caso 3:
                llenar campo[mediofila-1][mediocolumna]=  “region” + cantidadregiones
                llenar campo[mediofila][mediocolumna-1]=  “region” + cantidadregiones
                llenar campo[mediofila-1][mediocolumna-1]=  “region” + cantidadregiones

        aumentar la cantidad de regiones del campo en 1
```

3.4°) Teorema maestro y resolución ecuación recurrencia:

a)Teorema Maestro:

Ecuación de recurrencia = T(n)= 4 * T(n/4) + O(1)

Regla teorema maestro:

Si: logb(a)< C -> T(n)= O(n^C)
Si: logb(a)= C-> T(n)= O(n^C * logB(n))= O(n^C log(n))
Si: logb(a)> C-> T(n)= O(n^logB(A))

con A=4, B=4 y C=2, nos queda que la complejidad temporal es O(n^2)

b)Ecuación de recurrencia:

3.5°) Ejemplo de funcionamiento:

Se realiza un muestra paso a paso de cómo el algoritmo completa con L un ejemplo de n=4 y silo en (3,0):

(0;0)	(0;1)	(0;2)	(0;3)
(1;0)	(1;1)	(1;2)	(1;3)
(2;0)	(2;1)	(2;2)	(2;3)
(3;0)	(3;1)	(3;2)	(3;3)

En el primer llamado se envía que el lugar que debe quedar vacío es el (3;0) (Silo), tomamos la región superior izquierda de manera dinámica basándose en el comienzo y final de la matriz.
Como el campo que debe quedar vacío no se encuentra en este sector se le pide que se deje vacío el inferior derecho (1;1).

(0;0)	(0;1)
(1;0)	(1;1)

(0;2)	(0;3)
(1;2)	(1;3)

(2;0)	(2;1)
(3;0)	(3;1)

(2;2)	(2;3)
(3;2)	(3;3)

Como llegamos al caso base que la matriz es de 2x2 se completa evitando la región que se recibe de la recursión anterior.

(0;0)	(0;1)
(1;0)	(1;1)

(0;2)	(0;3)
(1;2)	(1;3)

(2;0)	(2;1)
(3;0)	(3;1)

(2;2)	(2;3)
(3;2)	(3;3)

Esta recursión termina aquí retornando a las secciones originales ahora prosigue con la superior derecha, como esta no tiene el silo, se le indica que deje vacío el sector inferior izq (1;2), y de la misma manera se rellena el resto.

(0;0)	(0;1)
(1;0)	(1;1)

(0;2)	(0;3)
(1;2)	(1;3)

(2;0)	(2;1)
(3;0)	(3;1)

(2;2)	(2;3)
(3;2)	(3;3)

Esta recursión termina aquí retornando a las secciones originales, ahora prosigue con la Inferior Izquierda, como esta tiene el silo, se le indica que lo deje vacío (3;0), dicho sector a diferencia de los otros la presencia del silo genera un cuadrado completo que cuando sea acoplado al resto, dejará espacios vacíos a su alrededor en forma de L esto se propaga por todos los sectores padres donde se encuentra el silo lo mismo que la generación de los espacios en blanco aledaños.

(0;0)	(0;1)
(1;0)	(1;1)

(0;2)	(0;3)
(1;2)	(1;3)

(2;0)	(2;1)
(3;0)	(3;1)

(2;2)	(2;3)
(3;2)	(3;3)

Esta recursión termina aquí retornando a las secciones originales ahora a diferencia de las anteriores como se detectó el silo en este sector se rellena los espacio aledaños a la esquina con otra L (1;1), (1;2) y (2;2).

(0;0)	(0;1)
(1;0)	(1;1)

(0;2)	(0;3)
(1;2)	(1;3)

(2;0)	(2;1)
(3;0)	(3;1)

(2;2)	(2;3)
(3;2)	(3;3)

Finalmente tomara el sector Inferior Izquierdo, como esta no tiene el silo, se le indica que deje vacío el sector superior izq (2;2), y de la misma manera se rellena el resto.

(0;0)	(0;1)
(1;0)	(1;1)

(0;2)	(0;3)
(1;2)	(1;3)

(2;0)	(2;1)
(3;0)	(3;1)

(2;2)	(2;3)
(3;2)	(3;3)

Dejando un total de 5 regiones

3.6°) Código:

El código se encuentra en el siguiente repositorio en la carpeta tp1/ej3 y se adjunta el .zip con el código fuente.
https://github.com/sebastian-outeiro/tda

Ejecución - JAR

Se adjunta en los archivos un JAR pre-compilado para su rápida ejecución. En caso de que su ejecución no funcione, se deberá proceder a compilar el mismo nuevamente.

1°) Ir a la carpeta:

/tp1/ej3

2°) Abrir una consola en la ubicación y ejecutar el comando

java -jar ej3.jar <tamaño campo> <columna silo> <fila silo>

3°) Ejemplos

java -jar ej3.jar 8 0 0

Compilación - JAR

En caso de que sea requerido re-compilar el JAR se pueden seguir los siguientes pasos:

1°) Ir a la carpeta:

/tp1/ej3

2°) Compilar con maven:

mvn compile

3°) Armar el package con maven

mvn package

4°) El .jar ejecutable se creará en la carpeta /target. Se podría ejecutar el mismo con el siguiente comando:

java -jar target/*.jar 8 0 0

3.7°) Análisis de complejidad expuesta vs real:

elementos (n)	real milisegundos
2	2
4	3
8	4
16	4
32	4
65	5
128	8
256	16
512	33
1024	92
2048	303
4096	903
8192	3377
16384	Out of memory

Para realizar este análisis, hicimos un cuadro comparativo, de cantidad de n contra los milisegundos que tarda por ejecución y se puede apreciar una curva similar a la que tendría una parábola, a medida que aumentan las muestras cada vez su pendiente es mayor, reforzando lo indicado en la sección 4 del ejercicio.