Teoría de Algoritmos (75.29 / 95.06)
Trabajo Práctico n.°1 (2da re-entrega)

Curso Víctor Podberezski - Grupo Matcheados por G-S

Integrantes

Bohórquez, Rubén	109442
Diem, Walter Gabriel	105618
Loscalzo, Melina	106571
La Torre, Gabriel	87796
Uribe, Mauricio	105971

Índice

Notas sobre la re-entrega

Las partes 1 “La Campaña” y 3 “La regionalización del campo” no fueron modificadas, solamente se trabajó sobre la parte 2 “Reunión de Camaradería”.

Se hizo un cambio de enfoque total respecto a cómo encaramos el problema, apalancando el problema de maximum independent set (MIS) y la posibilidad de reducir o entender el problema dado a ese. Resolviendo así el problema MIS con el algoritmo greedy para encontrar la solución al problema de invitaciones de empleados.

Se agregaron en la bibliografía las referencias y papers utilizados para entender mejor la naturaleza del problema MIS y la investigación sobre este problema con algoritmos greedy.

Parte 1: La Campaña

Descripción del problema

Un influencer tiene un *valor de penetración del mercado*. Cuanto mayor es, mayor posibilidad de que más seguidores se interesen en lo que venden, es un número entero positivo. Un influencer tiene una lista de otros influencers con los que no quiere trabajar. Esta información es conocida.

Teniendo un conjunto de influencers, se quiere seleccionar un subconjunto de los mismos de forma que se maximice el valor de penetración total (la suma de los valores de penetración de cada influencer).

Solución propuesta

Árbol de estados del problema

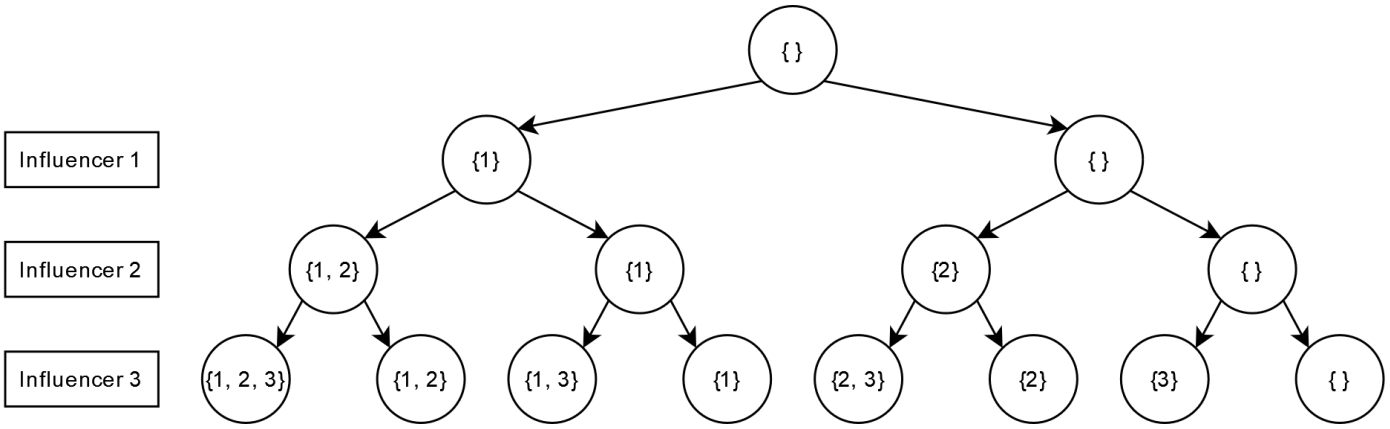
Sea *la lista inicial* la lista de tamaño n con los influencers a considerar, y sea *la lista resultado* la lista de influencers que consideramos como la solución óptima del problema. Cada influencer será representado con un id, y la lista está ordenada de mayor valor de penetración a menor.

Se define la *compatibilidad* de un influencer en relación a otros, como la característica de si puede trabajar con él o no. Si dos influencers no quieren o pueden trabajar juntos, son incompatibles. Las incompatibilidades son de pares, por lo que si el influencer A no trabaja con el B, la viceversa se cumple.

Se usará una estructura de árbol binario, con *n* niveles de profundidad, cada nivel corresponde a agregar el i-ésimo influencer a la lista resultado, con *i* entre 1 y *n*. La raíz corresponde a no tener ningún influencer agregado a la lista resultado.

Al ser un árbol binario, cada nodo tiene dos hijos, uno será agregar a la lista resultado actual el influencer correspondiente al nivel inferior (rama izquierda), y el otro será mantener la lista como está, o sea, no agregarlo (rama derecha).

Para un caso con 3 influencers, el árbol de estados, con la lista resultado en cada nodo, se vería de la siguiente manera:



Poda del árbol

La poda del árbol se hará en base a la compatibilidad. Dado un nodo en el nivel *i*, se podará la rama descendiente izquierda si el influencer del nivel *i+1* es incompatible con alguno de los de la lista resultado actual.

Función de costo L

Dado un determinado nodo en el nivel de profundidad *i* del árbol, que tiene descendientes, se tiene una lista resultado actual y un nivel *i+1* inferior.

Sea la función "valor" *v*, para un nodo es la suma de todos los valores de penetración del mercado de cada influencer de la lista resultado del nodo, para un influencer es simplemente su valor de penetración.

Vale la pena recordar que mientras más profundo está el nodo, menor es la penetración de mercado del influencer de ese nivel, dado el ordenamiento que se comentó anteriormente, esto es:

$$v_{Influencer\ i} \geq v_{Influencer\ i+1} \geq v_{Influencer\ i+2} \geq \ \ldots$$

Definimos la función costo como:

$$L_{nodo\ i+1} = v_{nodo\ i} + (n - i)v_{Influencer\ i+1}$$

Esto es, el costo del nodo *i+1* será el valor del nodo *i* (acumulación actual) sumado a la estimación de que en el mejor caso, los *n-i* nodos restantes también tendrán el valor de penetración de mercado del influencer *i+1*.

Recorrido del árbol

Para recorrer el árbol de estados utilizaremos el algoritmo Depth First Branch And Bound. Esto implica comenzar por la raíz con la lista resultado vacía. Dado un determinado nodo se verifica si la lista resultado es una solución mejor que la actual. Se expanden los descendientes, se poda de ser necesario y se calcula la función costo. Para los descendientes no recorridos y que puede resultar en una mejor solución, se selecciona al que tenga mayor función costo y se ejecuta de nuevo el proceso. Una vez que no quedan nodos descendientes para recorrer se vuelve al nodo ascendente inmediato.

Pseudocódigo

```
Sea influencer un tipo de dato que tiene un valor, un ID numérico y una lista de valores
numéricos que representan IDs de otros influencers
```

```
Sea listaInicial la lista de n influencers de la campaña, con n la cantidad total de influencers distintos
Sea listaActual una lista vacía de influencers
Sea valorListaActual el valor acumulado de los valores de penetración de cada influencer de la listaActual
Sea posInfluencerActual el influencer que se está considerando actualmente (nivel de profundidad del árbol)

ordenarLista(listaInicial)
listaActual= {}
valorListaActual = 0
mejorLista = {}
valorMejorLista = 0
posInfluencerActual = -1

ordenarLista(lista):
     Ordenar lista por el valor influencer.valor de mayor a menor con algoritmo quicksort.

armarNodo(posActual, listaActual, valorListaActual):
     posSiguiente = posActual + 1
     Sea influencerActual el influencer en la posición posActual de listaInicial. Si posActual es -1, influencerActual contiene un influencer vacío (0 de valor)
     Sea influencerSiguiente el influencer en la posición posSiguiente de listaInicial
     listaIzq = listaActual ∪ {influencerSiguiente}
     valorListaIzq = valorListaActual + influencerSiguiente.valor
     listaDer = listaActual
        valorListaDer = valorListaActual

     return {posActual, listaActual, valorListaActual, influencerActual, influencerSiguiente, listaIzq, valorListaIzq, listaDer, valorListaDer}

DepthFirstBranchAndBound(listaActual, valorListaActual, posInfluencerActual)

DepthFirstBranchAndBound(listaActual, valorListaActual, posActual):
     Si valorListaActual >= valorMejorLista:
            mejorLista = listaActual
            valorMejorLista = valorListaActual

     Si posActual == (n-1):
            return
     Sino:
            nodoActual = armarNodo(posActual, listaActual, valorListaActual)

            Si navegoIzquierda(nodoActual) == true:
                  DepthFirstBranchAndBound(nodoActual.listaIzq , nodoActual.valorListaIzq, posActual + 1)

            Si navegoDerecha(nodoActual) == true:
                  DepthFirstBranchAndBound(nodoActual.listaDer , nodoActual.valorListaDer, posActual + 1)
```

```
            return

esCompatible(influencerAVerificar, listaInfluencers):
      esCompatible = true
      Para influencer en listaInfluencers:
            Para idInfluencerIncompatible en influencer.incompatibilidades:
                  Si idInfluencerIncompatible == influencerAVerificar.id:
                        esCompatible = false
      return esCompatible


costo(posActual, valorListaActual, valorInfluencerSiguiente):
      return valorListaActual + (n - posActual) * valorInfluencerSiguiente

navegoIzquierda(nodo):
      costoIzq = costo(nodo.posActual, nodo.valorListaActual, nodo.influencerSiguiente.valor)
      Si esCompatible(nodo.influencerSiguiente, nodo.listaIzq) y costoIzq > valorMejorLista:
            return true
      Sino:
            return false

navegoDerecha(nodo):
      costoDer = costo(nodo.posActual, nodo.valorListaActual, nodo.influencerSiguiente.valor)
      Si esCompatible(nodo.influencerSiguiente, nodo.listaDer) y costoDer > valorMejorLista:
            return true
      Sino:
            return false
```

Análisis de complejidad

Pre-procesamiento

Para preparar todas las estructuras de datos, se tiene que leer la entrada de un archivo y ordenar a los influencers por orden de penetración de mercado. La longitud del archivo depende de la cantidad de influencers que contenga *al cuadrado* porque cada influencer podría tener $n - 1$ incompatibilidades, por lo que la lectura y carga en memoria tiene complejidad tanto espacial como temporal de $O(n^2)$. Para el ordenamiento, existen algoritmos conocidos de complejidad temporal $O(nlog(n))$ y complejidad espacial $O(1)$. También nos va a interesar que las incompatibilidades de los influencers estén en una lista. Las listas de incompatibilidad se consideran parte de la estructura de datos del influencer, por lo que no incurren en costo espacial que no haya sido ya considerado.

En total, la complejidad del pre-procesamiento va a ser $O(n^2 + nlog(n))$, o sea, $O(n^2)$.

Algoritmo: complejidad temporal

En el peor de los casos el algoritmo recorre todos los estados del árbol de estados (si la solución es el último estado a recorrer), que como es un árbol binario son en total 2^n. En cada estado, además, se tienen que calcular 3 cosas: función de costo, condición de poda, y estados siguientes. Los estados siguientes son 2: el que corresponde a agregar al influencer $i + 1$, y el que no, y ambos se pueden generar en $O(1)$.

La función de poda consiste en verificar si $i + 1$ se encuentra entre las incompatibilidades de algún influencer del estado actual (o si alguno de los influencers del estado actual se encuentra entre las incompatibilidades de $i + 1$, pero ambos procedimientos son equivalentes en complejidad), esto se puede hacer en $O(n)$ con búsqueda lineal. En el peor de los casos (último nivel), ese chequeo se hace contra todos los otros influencers, para una complejidad temporal de $O(n^2)$.

La función costo también se puede calcular en $O(1)$ con uso inteligente de estructuras de datos: todos los componentes del cálculo (nivel del árbol, cantidad de influencers, valor del estado actual, penetración del influencer siguiente) pueden ser almacenados con su estructura de datos respectivas y ser accedidos y actualizados en tiempo constante.

En total, por cada uno de los estados se realizan 3 operaciones, para una complejidad de $O(2^n \cdot (1 + n^2 + 1))$, que se simplifica a $O(2^n \cdot n^2)$.

Estructuras de datos: complejidad espacial

Para la solución del problema, se utilizan 2 estructuras de datos a tener en cuenta: *influencer* y *estado*.

Un influencer tiene una id y un valor de penetración (ambos constantes numéricas), así como una lista de las ids de todos los influencers con los que es incompatible. En el peor de los casos, todos los influencers son incompatibles con todos los otros, por lo que cada uno tiene una lista de tamaño n . Entonces, vamos a tener n influencers que ocupan a lo sumo n, para un total de $O(n^2)$ en memoria.

El estado del árbol almacena el valor total actual de la campaña, el "nivel" del árbol al que corresponde, y la lista de influencers que corresponden a esa campaña. La lista de influencers es a lo sumo de tamaño n (todos los influencers), y se puede almacenar nada más la id o un puntero de cada influencer. Además, como el árbol se genera dinámicamente y se descartan los estados ya explorados, en memoria nunca van a coexistir todos los estados. A lo sumo, si se está en el último nivel del árbol, van a estar en memoria todos los estados anteriores hasta la raíz, para un total de n (profundidad del árbol) estados de tamaño n**,** que ocupan a lo sumo $O(n^2)$ en memoria.

Por lo tanto, la complejidad espacial va a ser $O(n^2 + n^2)$ que se simplifica a $O(n^2)$.

Código de la propuesta

El código del algoritmo propuesto realizado en el lenguaje de programación Python se puede encontrar en el zip entregado, en particular en el archivo *ej_1.py*. Sólo requiere Python instalado, ningún paquete extra. Se puede correr por consola como muestra el enunciado, ejemplo: *python ./ej_1.py "./carpeta/archivo.txt"*. O también se puede iniciar el script sin argumentos y por entrada estándar escribir el nombre/ubicación del archivo. Se provee un archivo *ej_1_tests.py* que corre un set de pruebas sobre el módulo *ej_1.py*, utilizando archivos de prueba guardados en la carpeta *ej_1_test_files*. También se puede encontrar en el repositorio de github.

Comparación de complejidades teóricas y reales

La construcción del código se realizó respetando el pseudocódigo, adaptando la forma de programar la solución para tener la misma complejidad. Los métodos *built-in* de ordenamiento de listas en python son de la complejidad considerada.

Algunas notas respecto a algunas estructuras de Python:

- Para una lista, el método *append* tiene complejidad temporal $O(1)$.
- Para una lista, acceder a un elemento por índice tiene complejidad temporal $O(1)$.
- Para un diccionario, acceder a un key-value pair tiene complejidad temporal $O(1)$.

Ejemplo simple de resolución de una instancia

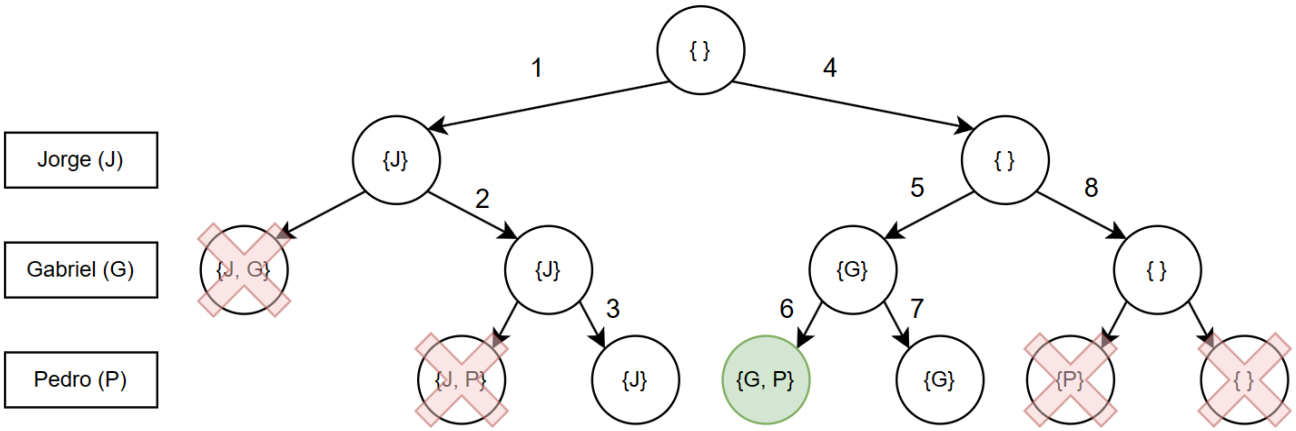
Como un ejemplo de cómo funciona el algoritmo, se presenta el contenido del archivo de texto con influencers que se usará:

```
1, Gabriel, 11, 2
2, Jorge,   15, 1, 3
3, Pedro,   5,  2
```

El resultado debería ser una lista con Gabriel y Pedro, en ese orden, con un valor de penetración acumulado de 16 puntos.

Se presenta un diagrama del árbol de estados para ilustrar el paso a paso del algoritmo, donde cada nivel del árbol tiene el nombre del influencer asociado, dentro del nodo se encuentra el estado actual de la lista, sobre las aristas recorridas hay un número indicando el orden en que fueron recorridas, y los nodos que no fueron recorridos ya sea por poda de incompatibilidad o por poda de la función costo, están marcadas con una cruz (X).

Una vez cargados los influencers, se procede a ordenar la lista de influencers por valor de penetración de mercado, de mayor a menor. Quedando la lista inicial {J,G,P}, usando las iniciales de los influencers para identificarlos. Con esto se sabe cuáles serán los influencers asociados a cada nivel del árbol, entonces ahora se procede a recorrer los estados acorde al diagrama.



PASO 1: Se navega a izquierda y se actualizan los valores de la lista resultado. Como es la primera navegación, esta es por defecto la mejor en este punto.

```
listaResultado = {J}
valorListaResultado = 15
```

PASO 2: A izquierda no se puede navegar porque hay incompatibilidad entre J y G, esa rama de estados queda podada. Entonces se navega a derecha. La lista resultado permanece igual porque no se agregó ni quitó nada.

```
listaResultado = {J}
valorListaResultado = 15
```

PASO 3: A izquierda no se puede navegar porque hay incompatibilidad entre J y P. Entonces se navega a derecha. La lista resultado permanece igual porque no se agregó ni quitó nada.

```
listaResultado = {J}
valorListaResultado = 15
```

PASO 4: Como se llegó al final del árbol se hace backtrack hasta el punto más alto donde se partió el flujo que aún no fue explorado, esto nos lleva al comienzo del árbol. Se navega a derecha porque ya se navegó a izquierda. La lista resultado permanece igual porque el nodo actual no contiene una mejor lista (está vacío).

```
listaResultado = {J}
valorListaResultado = 15
```

PASO 5: Se navega a izquierda. La lista resultado permanece igual porque el nodo actual no contiene una mejor lista, su valor es de 11.

```
listaResultado = {J}
valorListaResultado = 15
```

PASO 6: Se navega a izquierda. Se actualiza el valor de la lista resultado porque se encontró una lista con mayor valor.

```
listaResultado = {G,P}
valorListaResultado = 16
```

PASO 7: No queda más profundidad para recorrer, entonces se hace backtrack. Luego se navega a derecha. La lista resultado permanece igual porque el nodo actual no contiene una mejor lista, su valor es de 11.

```
listaResultado = {G,P}
valorListaResultado = 16
```

PASO 8: No queda más profundidad para recorrer, entonces se hace backtrack hasta el nivel de Jorge. Luego se navega a derecha. La lista resultado permanece igual porque el nodo actual no contiene una mejor lista (está vacía). Los estados descendientes de este quedan descartados y no recorridos porque resultan podados por la función costo, dado el valor de la lista resultado es 16, mayor que el costo de los descendientes directos que es 5, entonces no tiene sentido recorrerlos.

```
listaResultado = {G,P}
valorListaResultado = 16
```

Parte 2: Reunión de Camaradería

Descripción del problema

Se tiene un organigrama de una empresa que está representada en forma de árbol. Cada nodo es un empleado, sus hijos son sus subordinados inmediatos, y su padre es su jefe inmediato. Un empleado no puede tener más de un jefe.

Se quiere “invitar” a la mayor cantidad de empleados de la empresa, es decir, seleccionar la mayor cantidad de nodos del árbol, teniendo en cuenta la restricción de que si se invita a un jefe, sus subordinados inmediatos no podrán asistir. Esto evitaría problemas que sucedieron antes cuando sí se invitaron jefes y sus subordinados inmediatos.

Solución propuesta

Maximum Independent Set

El problema se puede interpretar como la búsqueda del set independiente máximo (Maximum Independent Set, o MIS) dentro del árbol organigrama.

Un set independiente de vértices es un conjunto de vértices en el cual todos ellos no comparten aristas entre sí. El problema MIS es encontrar el set independiente de mayor cardinalidad para un grafo dado. En nuestro caso sería para el árbol organigrama.

Se puede considerar que se puede reducir en tiempo polinomial el problema dado al problema MIS. Tomando el árbol organigrama como el grafo del MIS,y el set resultado del problema MIS como el set resultado de a quiénes se quiere invitar.

Elección Greedy

Para la elección greedy se utilizará la heurística de tomar el nodo del árbol de menor grado considerando únicamente aquellos nodos que aún puedan ser seleccionados, es decir, aquel que posea la menor cantidad de aristas que lo conectan con otros nodos que aún son candidatos a formar parte del set independiente. En términos del problema original, se selecciona al empleado con menos relaciones directas (empleado-jefe o jefe-empleado) con otros empleados que aún puedan ser invitados a la reunión (es decir, que no tienen relación directa con ninguno de los invitados hasta ese momento).

Subestructura del problema

Al elegir el nodo con el grado mínimo y descartar sus vecinos inmediatos, se está eligiendo aquel nodo que descarta la mínima cantidad de vecinos. Para un nodo dado del organigrama de la empresa, un nodo tiene como máximo un padre (el jefe de dicho empleado) y tiene una cantidad *m* de hijos (subordinados).

Elegir el nodo de menor grado y descartar los adyacentes involucraría que el empleado sea invitado, su jefe sería descartado y si el nodo tiene hijos, estos, o sea los subordinados, también serían descartados. De esta manera se cumpliría el objetivo de buscar la cantidad máxima de empleados a invitar, sin que coincidan en la fiesta un empleado con su jefe ni con sus subordinados. Es decir, la solución se construye navegando la subestructura óptima del problema.

Pseudocódigo

Acorde al enunciado, ya contamos con el árbol organigrama, así que se asume que ya lo tenemos listo para utilizar en el algoritmo. Esto quiere decir que tenemos un grafo *G(V, E)*. Un vértice o *nodo* tiene un campo identificador de sí mismo, el identificador de su jefe (se considera que el CEO no tiene jefe), una lista de los nodos que son sus hijos y la longitud de esta lista. Los nodos hoja tendrán la lista de hijos vacía.

```
{ identificador, jefe, cantidadHijos, hijos[ nodoHijo1, nodoHijo2, ... ] }
```

El identificador podría ser cualquier campo que identifique de manera unívoca a un empleado. Se procede a mostrar el pseudocódigo:

```
Sea G el grafo que representa el organigrama que está compuesto por una lista V de vértices y una lista E de aristas donde cada vértice V posee la estructura mencionada anteriormente
Sea n la cantidad de vertices del grafo
Sea invitados una lista
Sea candidatos una lista de tamaño n

candidatos = G.V

def grado(nodo):
    Si nodo.jefe == null:
           return nodo.cantidadHijos
    Sino:
           return nodo.cantidadHijos + 1

def ordenar_candidatos(lista):
```

```
      ordenar lista de candidatos por grado (cantidad de hijos + jefe si tiene),
con algún método O(1) espacial y O(nlogn) temporal

   Mientras n != 0:
       ordenar_candidatos(candidatos)
       nodoAInvitar = candidatos[0]
       invitados.agregar(nodoAInvitar)
       candidatos.eliminar(nodoAInvitar)
       n--
       Si nodoAInvitar.jefe != null:
              // eliminar jefe como candidato y de las conexiones que tiene
              candidatos.eliminar(nodoAInvitar.jefe)
              n--
              Si nodoAInvitar.jefe.jefe != null:
                     nodoAInvitar.jefe.jefe.hijos.eliminar(nodoAInvitar.jefe)
                     nodoAInvitar.jefe.jefe.cantidadHijos--
              Para hijoDelJefe en nodoAInvitar.jefe.hijos:
                     hijoDelJefe.jefe = null
       Para hijo en nodoAInvitar.hijos:
              // eliminar hijos como candidatos y de las conexiones que tienen
              Si hijo está en candidatos:
                     candidatos.eliminar(hijo)
                     n--
                     Para subHijo en hijo.hijos:
                            subHijo.jefe = null
```

La lista con los empleados elegidos estará en la variable `invitados`.

Justificación de greediness

Dado el algoritmo en forma de pseudocódigo y las explicaciones anteriores se tienen todas las piezas necesarias para justificar que la solución propuesta es un algoritmo greedy, y estas son la subestructura óptima y la elección greedy factible e irrevocable.

Demostración de optimalidad

Sea *v* un nodo que junto a sus adyacentes forma un clique. Sólo un nodo de este clique podrá ser parte de la solución, ya que si se incluye más de uno, habrán aristas compartidas y no sería un set independiente. Dado que *v* y sus adyacentes forman un clique, siempre podremos tomar a *v* en particular, ya que no habrá conflictos con nodos externos porque el sistema está inmerso en un clique. Tomando a *v* entonces, estamos haciendo la elección óptima dentro del clique.

Ahora, sea A un árbol, siempre las hojas forman un clique de tamaño 2 (la hoja y su padre), y además siempre las hojas son los nodos de grado mínimo.

El algoritmo va seleccionando las hojas y re-convirtiendo el árbol original en otros árboles más pequeños mediante la eliminación de nodos. Por lo tanto, siempre seleccionará las hojas de estos árboles nuevos, tomando una decisión óptima por lo explicado anteriormente. Por lo tanto el algoritmo devuelve una solución óptima al problema de MIS, resultando en una solución óptima al problema de invitar a la mayor cantidad de empleados posible cumpliendo las restricciones del problema.

Análisis de complejidad

Pre-procesamiento

Sean **n** la cantidad total de empleados + el CEO, es decir, n es la cantidad total de nodos del "árbol organigrama" de la empresa, de ahora en más referido sólo como "árbol".

Se considera pre-procesamiento del árbol a la carga de la lista con sus nodos, cada nodo con su respectivo identificador, jefe, cantidad de hijos y lista de hijos.

Recorrer el árbol para generar la lista *candidatos* tiene una complejidad de O(n) siendo que solo se necesita recorrer cada nodo una vez para agregarlo a la lista de *candidatos* y su complejidad espacial también será O(n).

Algoritmo: complejidad temporal

La función *grado* tiene una complejidad simple de O(1).

El método *ordenarListaDeEmpleados* es un ordenamiento que puede ser implementado de diversas formas en O(n log n).

Obtener el primer elemento de *candidatos* y agregarlo a *invitados* tiene complejidad temporal O(1).

Eliminar el *nodoAInvitar* de la lista de *candidatos* podría considerarse O(1) ya que sabemos que es el primer elemento de la lista, pero esto dependerá de qué lenguaje estemos utilizando, por ejemplo, en Python, las listas están implementadas como arrays dinámicos, así que eliminar el primer elemento obligaría a que, internamente, se reordene el resto de elementos para cubrir ese lugar. Así que podemos asumir que en el peor de los casos la complejidad temporal sería O(n).

Eliminar al jefe, por lo dicho anteriormente, también tendrá complejidad temporal O(n), más en este caso que no sabemos en qué posición de *candidatos* estará el jefe.

Eliminar al jefe como hijo de su respectivo jefe tendrá complejidad temporal O(n).

Eliminar a los hijos de *candidatos*, si están, será O(n).

Eliminar al jefe de los hijos de los hijos será O(n) la recorrida de los hijos y O(1) la eliminación de los hijos.

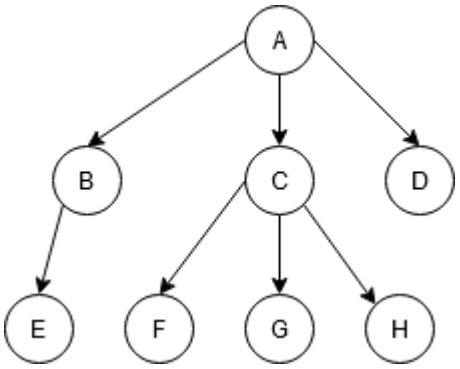
Entonces, omitiendo algunos O(1) obvios como la disminución de n en 1 unidad, dentro de cada bucle tendremos: O(n log n + 1 + 1 + n + n + n + n + n) = O(n log n). Ahora esto lo hacemos por cada elemento del árbol por lo que tendremos una complejidad temporal de **$O(n^2 \log n)$**.

Estructura de datos: complejidad espacial

La lista de *candidatos* tendrá un elemento por cada vértice con lo que su complejidad espacial será de O(n). La lista de *invitados* empieza vacía y su máxima longitud será de n, con lo cual su complejidad espacial será de O(n).

Por lo tanto la complejidad espacial total será de **O(n)**.

Ejemplo simple de una instancia del problema y su resolución



Paso 1: Obtener la lista de *candidatos*.

Posición	Id	Jefe	CantidadHijos	Hijos	Grado
1	A	-	3	B,C,D	3
2	B	A	1	E	2
3	C	A	3	F,G,H	4
4	D	A	0	-	1
5	E	B	0	-	1
6	F	C	0	-	1
7	G	C	0	-	1
8	H	C	0	-	1

Paso 2: Comienza el ciclo. Reordenar *candidatos* según grado.

Posición	Id	Jefe	CantidadHijos	Hijos	Grado
1	D	A	0	-	1
2	E	B	0	-	1
3	F	C	0	-	1
4	G	C	0	-	1
5	H	C	0	-	1
6	B	A	1	E	2
7	A	-	3	B,C,D	3
8	C	A	3	F,G,H	4

Paso 3: Agregar a la lista *invitados* el primer elemento de *candidatos*.
invitados = D.
Paso 4: Eliminar al jefe (A) del nodo invitado (D) de *candidatos*.

Posición	Id	Jefe	CantidadHijos	Hijos	Grado
1	E	B	0	-	1
2	F	C	0	-	1
3	G	C	0	-	1
4	H	C	0	-	1
5	B	A	1	E	2
6	C	A	3	F,G,H	4

Paso 5: Eliminar al jefe de los nodos hijos del jefe que se acaba de eliminar de *candidatos*.

Posición	Id	Jefe	CantidadHijos	Hijos	Grado
1	E	B	0	-	1
2	F	C	0	-	1
3	G	C	0	-	1
4	H	C	0	-	1
5	B	-	1	E	1
6	C	-	3	F,G,H	3

Paso 6: Como el nodo a invitar no tenía hijos, reordenar *candidatos* por grado, queda como en la figura anterior. Repetir el paso 3, agregar el primer elemento de *candidatos* a *invitados*.

invitados = D, E.

Paso 7: Eliminar el jefe (B) del nodo invitado (E) de *candidatos*.

Posición	Id	Jefe	CantidadHijos	Hijos	Grado
1	F	C	0	-	1
2	G	C	0	-	1
3	H	C	0	-	1
4	C	-	3	F,G,H	3

Paso 8: Eliminar al jefe (E) que se acaba de eliminar de *candidatos* de los hijos que tenía. Como se ve en la tabla del paso 5, esto no genera ningún cambio. Como E no tenía hijos, se repite el ciclo.

Paso 9: Agregar a la lista *invitados* el primer elemento de *candidatos*.

invitados = D, E, F.

Paso 10: Eliminar al jefe de F de la lista de *candidatos.*

Paso 11: Eliminar al jefe de F (C), de los hijos que lo tenían como jefe.

Posición	Id	Jefe	CantidadHijos	Hijos	Grado
1	G	-	0	-	0
2	H	-	0	-	0

Finalmente se repite el ciclo quedando:

invitados = D, E, F, G, H

Parte 3: La regionalización del campo

Descripción del problema

Se tiene un campo agrícola compuesto por cuadrados de una hectárea. Estos campos son cuadrados, con un lado teniendo *n* hectáreas, es decir que tiene área de *n·n*. Se puede interpretar al campo como una cuadrícula de dimensiones *n·n*, o sea, un cuadrado de dos dimensiones de lado *n*, compuesto por casillas de tamaño *1·1*.

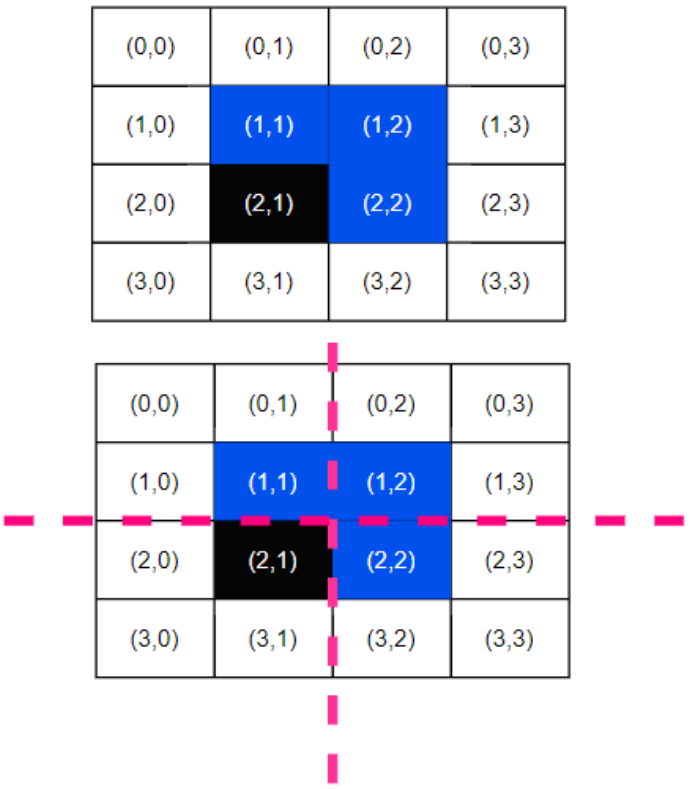
El número *n* es un número entero mayor o igual a 2 y es potencia de 2, por lo que puede tomar los valores 2, 4, 16, 32, etc. Se quiere encontrar regiones con forma de L simétrica (3 casillas que tomen la forma ┌, ┐, └ o ┘).

Se proporciona una imagen para ilustrar un estado resuelto y entender visualmente lo que se quiere buscar, donde se tiene una cuadrícula de *n = 4*, es decir de formato *4x4*, y 16 casillas con tamaño *1x1* cada una. Cada casilla contiene dentro su coordenada representada como un vector de dos dimensiones *(x,y)*, con el origen ubicado en la esquina superior izquierda. La casilla (2,1) es la que contiene los silos, así que queda descartada para el armado de regiones. Las regiones con forma de L se encuentran resaltadas con distintos colores.

(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)

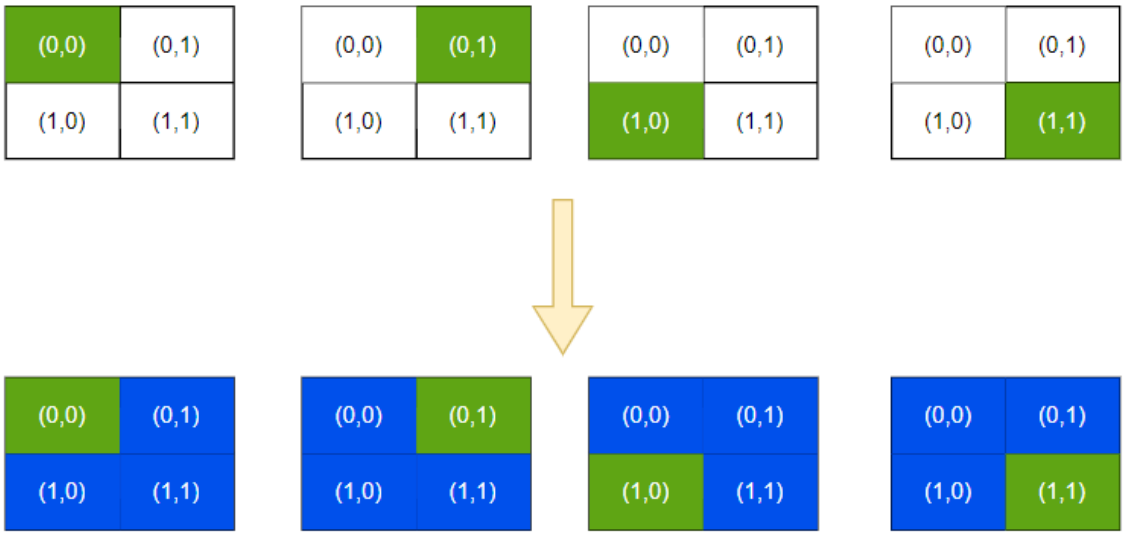
Solución propuesta

Para llevar a cabo la solución del problema planteado utilizaremos el método división y conquista. Nuestra idea es poder segmentar el área de cultivo en el cual posee dimensiones de $n \cdot n$, en cuatro secciones de tamaño $n/2 \cdot n/2$. Esta forma nos permitirá llegar a pequeños subcampos de tamaño 2x2.



Campo con dimensiones 2x2(n=2)

En este caso nuestro problema se reduce a un campo 2x2, donde una de las cuatro secciones es ocupada por los silos, es decir que no se puede cultivar. Las cuatro posibles ubicaciones para los silos son (0,0), (0,1), (1,0) y (1,1).

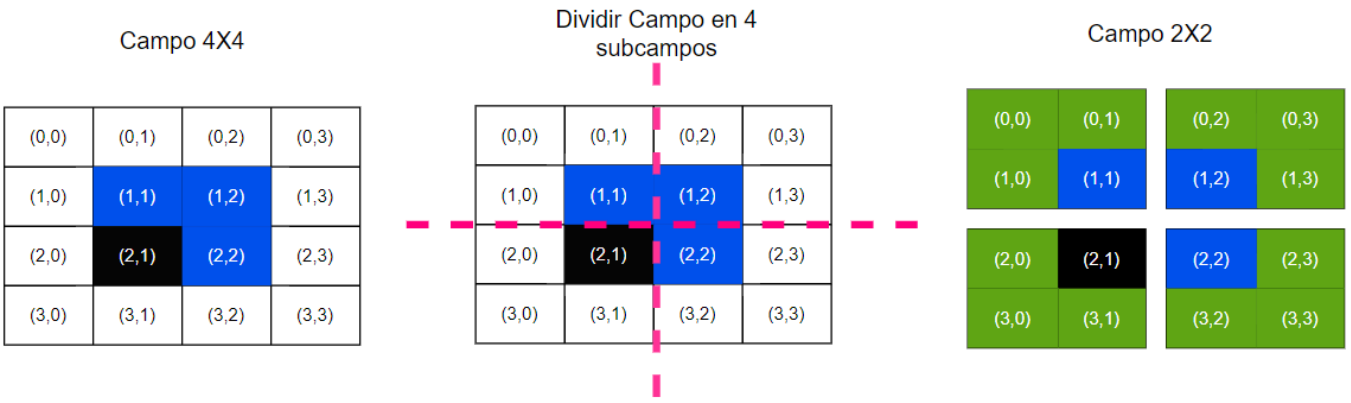


Podemos observar que al tomar esas posiciones las otras ubicaciones forman la “L” que permite cubrir las secciones restantes que sí se pueden cultivar. Por esta razón, la resolución del caso con dimensión 2x2 es directa ya que tratamos con un escenario mínimo y no necesitamos seguir subdividiendo.

Campo con dimensión: n>2

Ahora tenemos un campo que posee dimensiones más grandes teniendo en cuenta que n>2. Como vimos anteriormente, abordaremos el caso subdividiendo en cuatro secciones, de los cuales cada uno será de $n/2\ X\ n/2$ .De esta forma creamos nuevos subcampos que pueden reducirse al caso base de $n\ =\ 2$.

Primero se tiene que identificar el subsector que contiene los silos, y se sigue colocando las regiones para formar la “L” en el punto central donde se unen las demás secciones, de esta manera podemos crear un espacio ocupado que actúa como área no cultivable en cada uno. A continuación visualizamos, cómo se divide un campo en 4 secciones para llegar a subproblemas 2x2:



Por cada sección del campo original, si la sección contiene el silo, se va a “excluir” esa sección. Sino, se excluye la sección más cercana al centro. Luego cada una de esas secciones se subdividen y se repite el mismo procedimiento: se excluye la sección con el silo o la más cercana al centro de la sección que la contiene. Al llegar a las secciones de 2x2, se juntan las 3 secciones no “excluidas” en una L, y todas las secciones excluidas contiguas se juntan para formar otras L’s.

Relación de recurrencia

La relación de recurrencia del algoritmo está dada por la siguiente ecuación: $T(n) = 4 \cdot T(\frac{n}{2}) + O(1)$. Se realizan 4 llamados uno por cada cuadrante y el tamaño de los subproblemas es la mitad que del problema inicial y las demás operaciones son de tiempo constante.

Pseudocódigo

```
coordenadas(cuadrante):
    devolver las coordenadas de la esquina superior izquierda

cuadranteDelSilo(cuadrante, silo):
    Por subcuadrante del cuadrante:
        Si subcuadrante contiene al silo:
            devolver subcuadrante

// Todos los chequeos son a cantidades constantes (siempre hay 4 subcuadrantes), es O(1)
cuadranteCentral(subcuadrante, cuadrante):
    si subcuadrante tiene excluido:
        retornar
    si subcuadrante es el excluido de cuadrante:
        devolver subcuadrante correspondiente de subcuadrante
        // (i.e. si subcuadrante es el de abajo a la derecha,
        // devolver el de abajo a la derecha de subcuadrante)
    sino:
        x1, y1 = coordenadas(subcuadrante)
        x2, y2 = coordenadas(cuadrante)

        si x1=y2 e y1=y2:
            devolver inferior derecho de subcuadrante
        si x1=y2 e y1>y2:
            devolver superior derecho de subcuadrante
        si x1>y2 e y1=y2:
            devolver inferior izquierdo de subcuadrante
        si x1>y2 e y1>y2:
            devolver superior izquierdo de subcuadrante

Colorear(cuadrante):
    asigna un valor a las 3 casillas no excluidas y otro a la excluida

Seccionar(subcuadrante, cuadrante, silo):
    Si subcuadrante contiene al silo:
        cuadranteAExcluir = cuadranteDelSilo(subcuadrante, silo)
    Sino:
        cuadranteAExcluir = cuadranteCentral(subcuadrante, cuadrante)
```

```
    excluir(cuadranteAExcluir)

    Si tamaño(subcuadrante) == 2:
        colorear(subcuadrante)
    Sino:
        Por hijo en hijos(subcuadrante):
            Seccionar(hijo, subcuadrante, silo)

Sea campo el campo entero y silo las coordenadas del silo
Seccionar(campo, campo, silo)
```

Análisis de complejidad

Complejidad Temporal

Se presenta el análisis de complejidad temporal "desenrollando" la ecuación de recurrencia:

$$T(n) = 4T(\frac{n}{2}) + O(1)$$

$$T(\frac{n}{2}) = 4T(\frac{n}{4}) + O(1)$$

$$T(n) = 4^2T(\frac{n}{2^2}) + 4 \cdot O(1) + O(1)$$

$$T(\frac{n}{4}) = 4T(\frac{n}{8}) + O(1)$$

$$T(\frac{n}{4}) = 4^3T(\frac{n}{2^3}) + 4^2 \cdot O(1) + 4 \cdot O(1) + O(1)$$

$$T(n) = 4^iT(\frac{n}{2^i}) + \sum_{j=0}^{i} 4^j \cdot O(1)$$

con $T(2) = O(1)$ si $i = log_2(\frac{n}{2})$

$$T(n) = 4^{log_2(n/2)}T(\frac{n}{2^{log_2(n/2)}}) + (\frac{n}{2})^2 \cdot O(1) + \ldots + O(1) = (2^{log_2(n/2)})^2T(2) + (\frac{n}{2})^2 \cdot O(1)$$

$$T(n) = 2 \cdot (\frac{n}{2})^2 O(1) = O(n^2)$$

Ahora se presenta el análisis de complejidad temporal utilizando el teorema maestro:

$$A = 4,\ B = 2,\ C = 0 \Rightarrow log_B(A) > C \Rightarrow log_2(4) > 0$$

Por lo tanto, $T(n) = O(n^{log_B A}) \Rightarrow T(n) = O(n^2)$

Complejidad Espacial

El algoritmo presentado aborda la situación problemática operando sobre una matriz de tamaño *n·n*, que representa el campo. En el caso de dividir el área en subdivisiones, nuestro método trabaja sobre la misma matriz, es decir que opera in place. Además utiliza una matriz que ajusta su tamaño en base al proceso de subdivisión, por esta razón la complejidad espacial del algoritmo se mantiene en $O(n^2)$.

Código de la propuesta

El código se encuentra en el zip en el archivo *ej_2.py*. Sólo requiere Python instalado, ningún paquete extra. Se puede correr por consola como muestra el enunciado, ejemplo: *python ./ej_2.py <dimension> <x_silo> <y_silo>*. Por ejemplo se podría ejecutar como: *python ./ej_2.py 16 1 2*. También se puede encontrar en el repositorio de github https://github.com/gabrieldiem/tda-1-podbe-trabajo-practico-1.

Comparación de complejidades teóricas y reales

Toda la información necesaria de cada cuadrante (su dimensión, las coordenadas de su esquina superior izquierda, las coordenadas de sus subcuadrantes, y su subcuadrante excluído) se almacenan dentro de una estructura de datos, así que se pueden acceder en $O(1)$ como se había planteado en el pseudocódigo. Las funciones son prácticamente idénticas a las del pseudocódigo. La única notablemente distinta es la de `colorear` que parece recorrer una matriz, pero como siempre se llama sobre una sección de 2x2 en verdad siempre recorre 4 valores y por lo tanto es $O(1)$. En cuanto a la complejidad espacial, se almacena una matriz de n^2 con la respuesta. Además, cada campo contiene a su vez sus 4 subcampos, que haciendo la cuenta son n^2 campos en total, por lo que la complejidad espacial también se mantiene en n^2.

Corriendo un par de veces el programa para distintos tamaños se obtuvieron los siguientes resultados:

n	Tiempo promedio (s)	Uso de memoria prom. (Mb)
128	0.07	11.8
256	0.23	26.5
512	1.15	84.5
1024	5.11	323

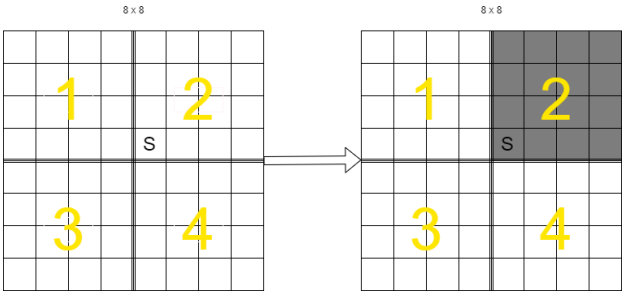
Como se puede ver, el tiempo de ejecución si crece de forma esperada, pero el uso de memoria es mucho menor al teórico (está entre n y $nlog(n)$ aproximadamente). Es posible que haya alguna optimización interna del lenguaje que reduzca el uso de memoria.

Cabe aclarar que para el análisis de complejidad no se tomó en cuenta la función de imprimir la respuesta por pantalla, ya que cumple fines meramente ilustrativos.

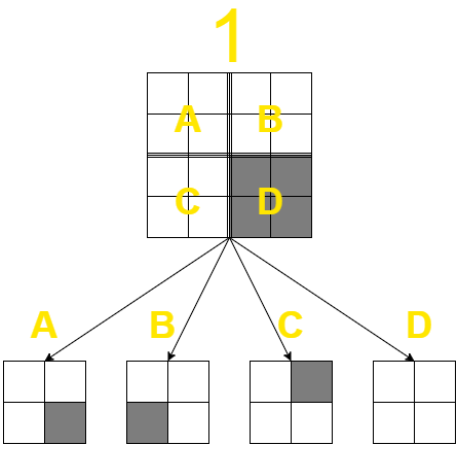
Ejemplo simple de resolución de una instancia

Inicialmente, se tiene un campo de 8x8 cuyo silo se encuentra en las coordenadas (y, x) = (3, 4). Luego, el algoritmo se encarga de verificar si el campo incluye al silo y como **lo incluye**, marca al cuadrante donde está el silo como **excluido** (el cuadrante “2” en gris).

Como no se encuentra en su caso base, el algoritmo sigue revisando recursivamente de la misma forma a cada cuadrante del campo.

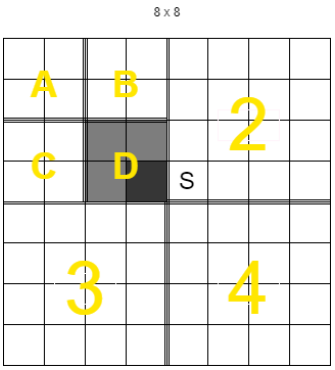


Luego, sigue el mismo proceso con cada cuadrante de 4x4. El primero es el cuadrante numerado como “1”. Como **no incluye** al silo, el algoritmo marca como **excluido** al cuadrante más cercano al centro del campo padre de 8x8. Es decir, el cuadrante “D”.



Continúa este proceso recursivamente con cada uno de los 4 cuadrantes de 2x2 “A”, “B”, “C”, “D” de “1”. Realizando la exclusión de los subcuadrantes más cercanos al centro en “A”, luego en “B” y en “C”.

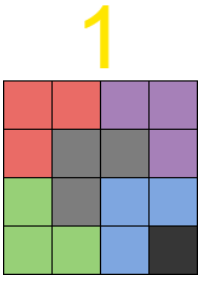
Al llegar al cuadrante “D” que era el cuadrante excluido de su cuadrante padre “1”, el algoritmo se ocupa de excluir al cuadrante hijo de “D” más cercano al centro del cuadrante padre de “D”. Esto se ha ilustrado en las siguientes imágenes con un gris más oscuro.



Además, los cuadrantes “A”, “B”, “C” y “D” tienen dimensión 2x2. Por lo que cumplen el caso base del algoritmo y serán pintados siguiendo las siguientes reglas:

1. Los cuadrantes no excluidos con un color.
2. El cuadrante excluido con otro color. Teniendo en cuenta que el color de los 3 excluidos de “A”, “B” y “C” tienen el mismo color.

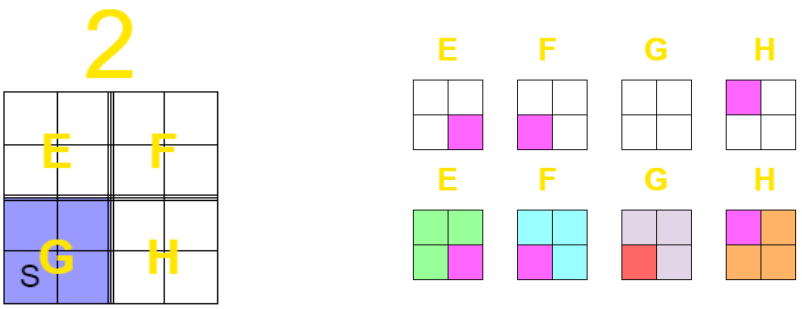
Una vez concluido el *coloreo,* se vuelve al *stack* dónde el siguiente llamado de `seccionar()` contempla al cuadrante “2”.



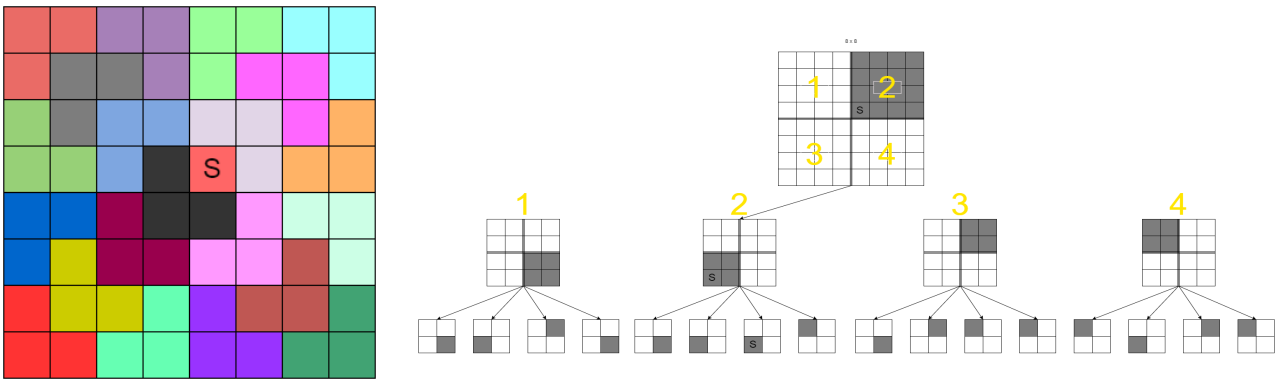
Cuadrante 2:

En este cuadrante de 4x4 **está incluido** el silo, por lo que se excluirá al cuadrante de 2x2 que lo incluye, el denominado “G”. Para luego `seccionar()` recursivamente a sus 4 cuadrantes “E”, “F”, “G”, “H”.

Comenzando por el cuadrante “E”, y ya que este **no incluye** al silo, realizará el mismo análisis anteriormente mencionado para excluir al cuadrante más cercano al centro. Esto se repetirá para el cuadrante “F” y “H” más adelante. El cuadrante “G” **incluye** al silo, por lo que se excluirá al cuadrante que incluye al silo. En este caso, es justamente el **cuadrante silo** el que se excluirá. Por lo que el coloreo resulta:



El resultado, luego de continuar con los llamados recursivos de los cuadrantes “3” y “4” siguiendo un esquema similar al visto para el cuadrante “1” es el siguiente:



Bibliografía

Sebastian Wild, IT University of Copenhagen, *"Quicksort, Timsort, Powersort: Algorithmic ideas, engineering tricks, and trivia behind CPython's new sorting algorithm*" <https://cs-lectures.itu.dk/lectures/231108-sebastian-wild-sorting-algorithms.html>

Python Official Wiki, *"TimeComplexity*" <https://wiki.python.org/moin/TimeComplexity>

Guy Blelloch, Jeremy Fineman, Julian Shun, *"Greedy Sequential Maximal Independent Set and Matching are Parallel on Average"* <https://arxiv.org/pdf/1202.3205>

Kevin Wayne, *"Extending the Limits of Tractability"* <https://cs.gmu.edu/~kosecka/cs483-001/10extending-approx.pdf>

Chandra Chekuri, *"Maximum Independent Set Problem in Graphs"* <https://courses.engr.illinois.edu/cs583/sp2016/LectureNotes/packing.pdf>

Mathieu Mari, *"Study of greedy algorithm for solving Maximum Independent Set problem"* <https://www.di.ens.fr/~mmari/content/papers/rapport.pdf>