ón para el mínimo coste de aspersores17Ejercicio 3equation.3.1 ón para ver si se va a llenar el terrenoón para ver si se va a llenar el terreno27Ejercicio 3equation.3.2 ón para ver si se puede llenar el terreno entre medio de los aspersores i,jón para ver si se puede llenar el terreno entre medio de los aspersores i,j37Ejercicio 3equation.3.3 ón para ver si con el aspersor actual se cubre el final del terrenoón para ver si con el aspersor actual se cubre el final del terreno47Ejercicio 3equation.3.4

DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP1: Técnicas Algorítmicas

21 de Septiembre de 2022 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Cappella Lewi, Federico Galileo	653/20	glewi@dc.uba.ar
Mallol, Martín Federico Alejandro	208/20	martinmallolcc@gmail.com
Teplizky, Gonzalo Hernán	201/20	gonza.tepl@gmail.com
Stemberg, Uriel Nicolás	213/20	uri.stemberg@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Resumen

Para este trabajo el enfoque será puesto en experimentar con distintas técnicas algoritmicas recursivas, con el fin de encontrar algoritmos más rápidos y eficientes que aquellos que emplean fuerza bruta. Mas allá de que en la mayoría de los casos sea imposible evitar una complejidad exponencial, se intentará acotar lo más posible el tiempo de cómputo de cada programa implementado en este trabajo. Se deben resolver tres ejercicios donde, cada uno de ellos se tratará sobre el comportamiento de una de estas tres técnicas en específico: backtracking, algoritmo goloso/greedy/miope (cualquiera de estas tres definiciones es válida), y por último, programación dinámica.

En cuanto a los primeros dos ejercicios de este trabajo, el desempeño de las implementaciones será testeada por un *juez online*[1].

El informe se divide entre los tres ejercicios. La estructura de cada ejercicio es la siguiente:

- **Introducción**: Se da un vistazo inicial del problema a resolver y la técnica algorítmica a utilizar. Luego se explica brevemente que caminos se tomaron y que experimentación se llevó a cabo (y con qué fines).
- **Desarrollo**: Se explica con qué herramientas fue realizado el ejercicio en particular y de qué trata cada archivo donde fue realizada la implementación de los algoritmos.
- **Experimentación, resultados y discusión**: Se visitan las técnicas de backtracking, algoritmos greedy, y programación dinámica. Se da un pantallazo inicial sobre cómo es el rendimiento de una implementación sin podas, con podas (tanto de restricción como de optimalidad), y con memoización en los casos donde se haya incluído.

  También se incluyen aquellas hipótesis que fueron confirmadas, o rechazadas, por los resultados que arrojaron los programas, y su consecuencia en la discusión que conllevó en el grupo. Hubo pasos en falso como tambien aciertos. No siempre la implementación de una poda o la memoizacion con cierta estructura tiene por qué mejorar drásticamente el rendimiento de un algoritmo.

  Luego de conocer cómo se comportan las distintas variantes de los algoritmos implementados en cada ejercicio, se pone el foco en los tiempos de computos que dichas variantes arrojan y las comparamos entre sí para convalidar ciertas suposiciones.
- **Conclusiones**: Se concluye que...

  Los valores ideales para los métodos fueron los siguientes:
  Ejercicio 1: **blablabla.**
  Ejercicio 2: **blablabla.**
  Ejercicio 3: **blablabla.**

Palabras clave: *Fuerza Bruta*, *Backtracking*, *Greedy*, *Programación Dinámica*, *Complejidad Temporal*, *Complejidad Espacial*.

Índice

[1] `https://onlinejudge.org/`

1. Ejercicio 1

Como primer consigna, se nos plantea resolver el problema 1098 de UVA, *Robots On Ice*.

Tenemos como dato las dimensiones de una grilla de n x m, con n,m naturales pertenecientes al intervalo cerrado [2,8], junto a tres posiciones que establecemos como puntos de *'check-in'* dentro de nuestro mapa.

Se espera que la grilla mencionada sea recorrida de forma tal que permita atravesar estos puntos al llegar a un cuarto, la mitad y las tres cuartas partes del recorrido a realizar, respectivamente.

En particular, el camino se hará comenzando por el *extremo inferior izquierdo*, más precisamente en la posición (0,0) de nuestro mapa, debiendo finalizar el recorrido en la posición a su derecha, la (0,1), pasando exactamente una vez por cada posición, incluido cada uno de los check-ins, en el momento que corresponde.

Para realizar este recorrido, podremos movernos a cada uno de los puntos de alrededor, es decir, hacia abajo, arriba, a la derecha e izquierda. Nuestro objetivo es, a partir de todos los datos y las premisas que tenemos a disposición, hallar **cuantos recorridos válidos podemos realizar**.

La resolución de este problema la llevaremos adelante usando la técnica algorítmica de Backtracking, implementación de fuerza bruta en la que buscaremos optimizar el algoritmo de forma tal que la cantidad y el tamaño de las entradas a utilizar pueda ser mayor que al no aplicar ningún tipo de optimización en particular, en pos de que el algoritmo termine para entradas más grandes.

Retomando la definición mencionada sobre el tablero y las posiciones recibidas, tenemos que por cada posicion hay cuatro direcciones posibles a las cuales podríamos eventualmente movernos. Esto nos da un total de 4^{n*m} mapas posibles, donde en realidad en vez de mapas estamos representando los movimientos que se tienen de una posicion (i,j) a otra similar vecina. Es decir, la **complejidad de una implementación por fuerza bruta va a ser de O(4^{n*m})**. Siempre podemos movernos únicamente un paso por vez, y no es posible moverse en diagonal, por lo que los movimientos se reducen a los 4 ya mencionados.

Pero notemos que todas estas combinaciones posibles de movimientos no representan efectivamente un mapa buscado. Para empezar, una vez que nos movemos de un casillero a otro, no esperamos volver sobre nuestros pasos. Es decir, se pretende recorrer cada posición del tablero una única vez, lo cual nos da la pauta de que es necesario llevar un registro de las posiciones recorridas. De este modo, parados en una posición, debemos poder saber si es posible movernos, y en ese caso, a dónde. Esto puede ser efectivamente realizable sólo si la posición vecina no fue visitada, y si el movimiento que se pretende hacer no excede los bordes del tablero, en principio.

Esto último nos ofrece una nueva etapa de control. Si estoy en un determinado borde de la grilla, tengo que considerar solamente aquellas posiciones relativas a las que puedo moverme desde donde estoy, de forma tal que dichas posiciones pertenezcan al tablero. Estas consideraciones son el primer paso hacia una resolución de BT en la que **aplicaremos podas** en pos de reducir el scope de soluciones candidatas-no válidas.

En este camino a la aplicación de las podas, en el marco de la extensión de las soluciones parciales, con la meta de poder cortar toda rama del árbol que sí estaría considerada por una fuerza bruta, comenzamos determinando que para aquellos caminos que logren llegar hasta el final del mapa sin haber "retrocedido" o sido podados, es decir, los que lleguen al paso n*m en la posición (0,1), serán solución válida que sumará al contador de caminos a devolver como resultado. Caso contrario, serán soluciones parciales que tratarán de extenderse generando 4 nuevos mapas: aquellos resultantes de moverse hacia cualquiera de los 4 puntos, a priori, posibles.

Decimos que a priori posibles ya que la imposibilidad de moverse hacia alguno o incluso a todos

los puntos de alrededor, constituirán la primer poda. Implementamos la idea de tener un mapa como una estructura que podría llegar o no a modificarse en cada recursión, pudiendo cambiar en cada paso, lo cual dependerá de si podemos avanzar en alguna dirección, algo que resolvimos preguntándonos por cada una:

-**¿Está en rango?** (Posición válida de la grilla de n*m)
-**¿Está ocupada?** Para ver esto, registramos nuestros pasos efectivos de modo tal que armamos una función que verifica con un hash.
-**¿Está bloqueada?** Consiste en ver que avanzar hacia la posición en cuestión, no nos va a dejar encerrados luego, obligándonos a volver sobre nuestros pasos.
-**¿Divide en dos el mapa?**

Aquí ya estamos podando, y si salimos airosos de esta verificación para movernos, entonces generamos el nuevo mapa, contando un paso más, y con la nueva posición ocupada. Puede resultar algo trivial lo hecho, aunque necesario, pero lo que realmente requerimos para dejar armado un algoritmo cuya terminación pueda ser rápida para inputs razonables, es otro tipo de podas, algo más "valientes". Sabemos que este es un ejercicio que nos plantea objetivos parciales, que son los puntos de *check-in*. Estos son los principales elementos para evaluar el mapa que hemos construido.

De ahí es que nos surgieron tres nuevas preguntas:

-¿Si el paso al que llegamos superó el valor de algún *check-in*, a éste ya lo atravesamos?
-¿Si el paso al que llegamos no alcanzó el valor de algún *check-in*, a éste aún no lo atravesamos?
-Es más. Para cada punto de *check-in* no atravesado. La distancia a la que estamos de éste, ¿es menor o igual a la cantidad de pasos que a lo sumo podríamos realizar hasta tener que arribar a él?

Una vez nos hicimos estas preguntas, pudimos llegar a las podas finales con las cuales convertimos al algoritmo en algo razonable. Con razonable, *veáse* que anterior a estas podas, el simulador en el Juez no finalizaba en el tiempo pedido.

La primera de las dos podas es por **factibilidad**: si ya tendríamos que haber pasado por algún *check-in* dado el número que posee, y en el paso actual no lo hicimos aún, ya estamos ante un camino incorrecto. Lo mismo aplica al revés. Si para el momento en el que evaluamos hemos pasado por un *check-in* cuyo número es mayor al del paso actual, también incumplimos la consigna.

La segunda poda es por **optimalidad**: si bien puede parecer que estamos por buen camino porque no pasamos por un *check-in* en el momento incorrecto, ni nos olvidamos a alguno por el mapa, si el próximo punto de control por el que tenemos que pasar está a mayor *distancia Manhattan o Euclidiana*[2] que la cantidad de pasos que podemos dar aún hasta llegar a éste, es que tampoco llegaremos a buen puerto con el mapa actual.

Aplicado esto, logramos resolver eficientemente el problema y pasar los tests del Juez.

En cuanto a tests unitarios:
Además, pensamos una resolución agregándole Programación Dinámica al problema, proponiendo una codificación especial de las posiciones:

[2] `https://es.wikipedia.org/wiki/Distancia_euclidiana`

2. Ejercicio 2

En este segundo ejercicio se nos presenta el problema 10382 de UVA, *Watering Grass*.

Se nos provee como dato un n correspondiente a la cantidad de aspersores instalados en un terreno rectangular, el cual mide l * w, siendo l el largo, y w el ancho del mismo. Además, cada uno de los aspersores en cuestión tendrá un radio r y una distancia desde el extremo izquierdo del rectángulo, ubicándose cada uno en el centro horizontal del mismo.

Nuestro objetivo es, a partir de los inputs mencionados, encontrar **la mínima cantidad de aspersores** que nos cubran todo el rectángulo.

En este caso, procedimos a resolver el ejercicio bajo la técnica Golosa o Greedy, donde vamos a querer construir un procedimiento heurístico que pueda funcionar como algoritmo que efectivamente resuelva este problema. Para esto, tuvimos que comprender cuál era cada una de las decisiones golosas que debíamos tomar, es decir, la mejor en cada punto, y en base a eso, utilizar los datos del input a nuestro favor para llevar estas decisiones adelante. A continuación, detallaremos como llegamos a la solución, y luego daremos una muestra de la complejidad y la correctitud del algoritmo.

En las primeras interacciones con el problema llegamos a complicarnos, más que nada con las posiciones de los aspersores, pero luego de razonarlo, lo que pensamos y decidimos usar para la implementación, es el hecho de considerar que del rango de extensión de cada aspersor, realmente lo que nos interesa es todo el espacio de intersección que tiene con el rectángulo. Es decir, que si bien vemos al aspersor como una figura circular de radio r, nos iba a alcanzar con quedarnos con el punto de la recta horizontal desde el que empieza a ocupar hasta el que termina, lo cual representa al pedazo del rectángulo que cubre cada aspersor.

Obviamente este pedazo va a ser mayor en la medida que el radio del circulo lo sea y se encuentre en una zona razonable dentro de los límites del rectángulo. Bajo esta idea, hay un tipo de aspersores que definitivamente no nos iban a servir, y a los cuales descartamos: aquellos cuyo diámetro no llega a alcanzar siquiera al ancho del rectángulo *(2 * r < w)*, dado que no nos servía para nada ese cubrimiento parcial que no era ni del ancho w. Para todo aspersor que pasara por esta primera evaluación, buscamos la forma de ir de los datos provistos de cada uno de ellos, a calcular **el límite izquierdo y derecho de cada uno** dentro del rectángulo, de modo de conocer efectivamente cuánto se extienden. Quienes maximicen las diferencias entre los valores de ambos extremos, constituirán aspersores candidatos a formar parte de la solución óptima.

Con esto, planteamos la estructura de cada **subproblema** y **decisión golosa** i a resolver:

-*Subproblema* i: Hallar la mínima cantidad de aspersores requeridos con el que podamos cubrir el rectángulo de forma tal que lo hagamos desde el extremo i (visto horizontalmente) hasta el final del rectángulo.

-*Decisión golosa* i: De todos los aspersores cuyo límite izquierdo sea menor o igual a i, es decir, que empiecen en i, que es hasta donde tenemos cubierto, o antes, tomamos el aspersor que sea de máximo cubrimiento, lo cual según definimos será aquel que maximice **limiteDer(aspersor) - limiteIzq(aspersor)**.

Con esto, podíamos directamente crear un algoritmo que aplique esta estrategia, del cual estábamos seguros que si intempestivamente tomábamos todo aspersor que más cubra, lo estaríamos haciendo con la mínima cantidad y seguro que llegaríamos al mismo valor que el de la solución óptima, teniendo en nuestras manos un algoritmo goloso. Pero, pensamos... ¿se podía hacer mejor que eso? La respuesta es sí.

Sucede que hacerlo así de naive, nos aumenta la complejidad en peor caso, ya que eventualmente podríamos estar recorriendo más de una vez los mismos aspersores. Puede darse, por ejemplo, de tener que cada uno de los n aspersores ocupaba una porción igual del largo l y de forma consecutiva de modo que requeríamos de los n aspersores para cubrir el l del rectángulo, teniendo una **complejidad de O(n*l)**.

Habiendo notado esto, vimos que si teníamos ordenados los aspersores por el límite izquierdo desde donde ocupan, entonces los analizaríamos consecutivamente según el lugar desde donde empiezan y sabemos que a lo sumo pasaríamos por ellos una sola vez. Para implementar esto, utilizamos una *cola de prioridad* como estructura de datos, la cual será llenada por aspersores cuya máxima prioridad se les asignará a los elementos de menor límite izquierdo dentro del pedazo de rectángulo que ocupan.
De esta forma, en cada decisión golosa tomada para resolver cada subproblema i con i entre [1,n], nos manejaríamos solo dentro del rango de aspersores que no empiezan más alla del punto hasta el que llevamos cubierto, y nos quedaríamos con **el que más se extiende a la derecha**.
Tomada la decisión, pudiendo haber o no terminado de cubrir todo el rectángulo en ese punto, sumaríamos un aspersor a la solución, y además, resolveríamos el subproblema *i+1* considerando desde el primer elemento que no era candidato para el subproblema i. Con esto hecho, la complejidad del algoritmo goloso sería de **O(n)**, lineal en la cantidad de aspersores, aunque tendría un costo algo mayor si tomamos el costo del ordenamiento de la estructura que almacena los límites izquierdo y derecho de los aspersores.

Un posible pseudocódigo que nos sirvió para entender el problema de esta forma, fue algo como:

Algorithm 1 wateringGrass(A):

```
ordenarAspersores(A)
i ← 0
cubierto ← 0
minAspersores ← 0
while i < |A| ∧ cubierto < w do
    cubierto ← aspersor de cubrimiento máx con extremo izq ≤ cubierto
    i ← índice de primer aspersor con extremo izq > cubierto
    minAspersores ← minAspersores + 1
end while
return minAspersores
```

Se ve que efectivamente se lleva adelante a lo sumo una pasada por cada aspersor, por lo que si preponderamos el costo de ordenamiento, podríamos resolverlo en **O(n * log n)**.

Nos queda, por último, probar la correctitud del algoritmo. Por la forma en la que trabajamos con este tipo de ejercicios, donde el peso se pone en gran parte sobre la demostración, nos propusimos a ver que:

- Si tengo una solución optima puedo modificarla para que use una elección golosa (1).

- Si tengo una secuencia de k decisiones golosas, puedo extenderlas para llegar a una óptima (2).

Lo que queremos con esto, es probar que el algoritmo goloso propuesto produce una solución óptima.

Comenzamos probando **(1)**: Queremos ver que **toda solución óptima para este problema es posible modificarla utilizando elecciones golosas**. Sabemos que existe la óptima, pero queremos aquella que use la golosa. Esto lo podemos demostrar de forma directa.

Tomamos $R_k = r_1$,.....,r_k una solución óptima del subproblema i, es decir, aquella que minimiza la cantidad de aspersores necesarios para cubrir desde el punto i hasta el final del rectángulo, y un $r \in \mathrm{R}_k$ tal que r representa un intervalo que contiene al i. Seguro que existe un r de estas características ya que no sería óptima la solución si hubiera algún pedazo del rectángulo no cubierto.

Como también habíamos mencionado, una decisión golosa en ese punto para resolver el subproblema i, será tomar el aspersor de mayor extensión que cubra desde i o antes al rectángulo. Si llamamos a ese aspersor G_i, y nuestro objetivo es poder introducirlo dentro de la solución óptima, lo que podemos hacer es, partiendo de R_k, crear $R'_k = R_k \cup \mathrm{G}_i$ - r.

Sabiendo que el aspersor G_i es el de máximo cubrimiento que pasa por i, entonces seguro que su cubrimiento sera $\geq$ al cubrimiento de r, además de que, considerando que lo que hicimos fue básicamente reemplazar un aspersor por otro, seguiremos usando la mínima cantidad posible y cubriendo todo el rectángulo. Por lo tanto, R'_k es óptima utilizando una elección golosa, y en general, podremos fabricarnos todas las R''_k que deseemos reemplazando en cada subintervalo por la decisión golosa en ese punto, manteniéndonos en una solución óptima. □

Por último, probemos **(2)**: queremos ver que **luego de tomar *k* decisiones golosas** G_k**,** $\forall k > 0$**, se puede extender hacia una solución óptima.** Veámoslo por inducción en k.

P(k): G_k se puede extender a una solución óptima $\forall k > 0$.

Caso base con k = 0: Como $G_0 = \emptyset$, es decir, aún no hemos tomado ninguna decisión golosa, luego si existe una solución óptima, lo voy a poder extender a ella.

Paso inductivo: queremos ver que si vale P(k), lo hace también P(k+1). Nuestra HI es que tomadas k decisiones golosas, podemos extendernos hacia una sol. óptima, y queremos saber si vale para k+1 decisiones golosas.

Sabemos que a medida que vamos tomando estas decisiones, el subproblema restante se hace cada vez más chico. Tomadas k elecciones golosas, y con el rectángulo cubierto hasta por ejemplo, k, nos queda por resolver el subproblema *k+1*, que encuentre los aspersores mínimos necesarios para cubrir el pedazo de rectángulo que se extiende desde k hasta el final del mismo.

Si **por HI** sabemos que existe un γ tal que $G_1 \cup$.....$\cup \mathrm{G}_k \cup \gamma$ cubren todo el rectángulo, luego γ es óptima para el subproblema *k+1*. Pero adicionalmente, por lo demostrado en **(1)**, toda solución óptima puede modificarse con elecciones golosas.

Por lo tanto, con el mismo cubrimiento y la misma cantidad de aspersores utilizados, tendremos también una solución óptima que utilice la decisión golosa G_{k+1}. Esto quiere decir que habrá un γ' que asegure que $G_1 \cup$.....$\cup \mathrm{G}_k \cup \mathrm{G}_{k+1} \cup \gamma'$ cubren todo el rectángulo usando la mínima cantidad de aspersores y por lo tanto, γ' es óptima para el subproblema *k+2*, probando así que G_{k+1} se puede también extender a una solución óptima, **como queríamos ver**. □

3. Ejercicio 3

Por último, en el tercer ejercicio se debe trabajar con una variante del anterior.

Vamos a contar con un terreno de $l \cdot w$, siendo nuevamente l el *largo* y w el *ancho*, pero ahora el planteo del problema es un poco distinto. Se vuelve a tener a disposición una cantidad n de aspersores con su *radio*"r" y su *posición*"pos", pero se les agrega un atributo más, su *costo*"c". Aquí radica la escencia del problema, ya que en vez de tener que encontrar la menor cantidad posible de aspersores que puedan regar todo el terreno, se debe encontrar el conjunto de aspersores que pueda **cubrir todo el terreno** y que **su costo total sea el mínimo posible** (entre todos los conjuntos de aspersores que pueden cubrir el jardín). Como el algoritmo que emplearemos arroja el mínimo costo posible y no el conjunto de aspersores, si hay más de un conjunto de aspersores que cumplan con el mínimo costo no altera el resultado final de la función.

Para resolver este problema en particular, se tomó como técnica algorítmica la **"Programación Dinámica"**, porque se buscó evitar que la función que soluciona este ejercicio no resuelva el mismo problema varias veces. Si esto ocurriera, se perdería mucho tiempo ejecutando cosas a las cuales ya se le conocen su valor.

A la hora de plantear el problema, tomamos un camino distinto al de la *resolucióngolosa* anteriormente mencionada. En vez de tener una **cola de prioridad** donde los aspersores están ordenados de mayor a menor según su prioridad (a menor límite izquierdo, mayor la prioridad), empleamos un **vector** para posicionarlos. Primero cargamos a los aspersores en este en base a su posición de llegada (es decir, el primero en ser cargado vía consola irá a la primer posicion, el segundo a la segunda, y así hasta el aspersor número "n"). Una vez que el vector estuvo listo, se lo ordenó con *mergesort* (coste $\mathbf{O(n \cdot log\ n)}$) nuevamente según su límite izquierdo en el jardín. La única diferencia es que ahora, para poder acceder a un aspersor, podemos hacerlo con su índice en $\mathbf{O(1)}$ sin alterar las propiedades del vector, mientras que con la cola de prioridad, sí o sí deberíamos desencolar sus elementos para llegar a determinada posición i, con i $\in \{1, ..., |cola|\}$.

Con nuestro conjunto ordenado de aspersores ya conocido, llamémoslo $\mathbb{A}$, donde a_i es el *iésimo* aspersor, especificamos el problema. Se busca encontrar la solución X tal que $X \subset \mathbb{A}$ y X riega todo el jardín y $\forall Y \subset \mathbb{A}$ que riegue todo el jardín, $costeTotal(Y) = \sum_{i=1}^{|Y|} costo(y_i) \geq \sum_{i=1}^{|X|} costo(x_i) = costeTotal(X)$. Veamos cómo es la función que resuelve esta problemática.

Se llama **f(i,j) = costeTotal(X)**. Tener en cuenta que, para todas las funciones a continuación con los parametros i, j, estos corresponden al *iésimo* y *jésimo* aspersor respectivamente. Y además, el $aspersor_i$ siempre está **a la derecha** del $aspersor_j$. Con esto en mente, la función principal (junto con sus auxiliares) es la siguiente:

$$f(i,j) = \begin{cases} 0 & \text{si } estaLleno(j) \\ -1 & \text{si } noSeLlenara(i,j) \\ f(i+1,j) & \text{si } noSeLlenara(i+1,i) \\ f(i+1,i) + costo(i) & \text{si } noSeLlenara(i+1,j) \\ min((f(i+1,j), f(i+1,i) + costo(i))) & cc \end{cases} \quad (1)$$

$$noSeLlenara(i,j) = \begin{cases} True & \text{si } (i = n \wedge \neg estaLleno(j)) \vee noPuedoLlenarlo(i,j)) \\ False & \text{si } estaLleno(j) \\ noSeLlenara(i+1,j) \vee noSeLlenara(i+1,i) & \text{cc} \end{cases} \quad (2)$$

$$noPuedoLlenarlo(i,j) = limiteDerecho(aspersores_j) < limiteIzquierdo(aspersores_i). \quad (3)$$

$$estaLleno(j) = (l = 0)((j \neq -1) \wedge \neg(limiteDerecho(aspersores_j) < l) \quad (4)$$

La función se resuelve con **f(0,-1)** y devuelve el mínimo coste de un conjunto de aspersores que pertenece a todos aquellos conjuntos de aspersores que riegan completamente el terreno. Es decir, si $\mathbb{O} = \{o_1, ..., o_m\}$ la solución óptima, siendo $\mathbb{S}$ el conjunto de soluciones posibles, entonces se tiene $\mathbb{O} \in \mathbb{S}$ / $f(0,-1) = \sum_{i=1}^{|\mathbb{O}|} costo(o_i)$.

Esta funcion tiene este comportamiento ya que existen dos casos, uno en el que el aspersor actual a_i forma parte de un conjunto óptimo $\mathbb{O}'$ de r elementos, y otro donde no lo hace. Si forma parte, la solución al problema será $(\sum_{i=1}^{r-1} costo(o'_k)) + costo(a_i)$. Mientras que si no lo hace, entonces el costo de la solucion optima $\mathbb{O}'$ de r elementos es $\sum_{i=1}^{r} costo(o'_k)$. Siempre y cuando exista ese óptimo, lo elegiremos, mientras que si no hay solución al problema, es decir, el conjunto solución $\mathbb{S}$ está vacío, se devolverá -1.

Ahora bien, **¿vale la pena memoizar?** Si la función es correcta se puede simplemente programar y listo. El problema ya se resuelve. Sin embargo, el inconveniente que se presenta es que existe la superposición de subproblemas en este algoritmo, ya que calculamos muchas veces instancias ya resueltas. Por esto mismo, empleamos **Programación Dinámica**. Esto se debe a que la cantidad de llamados recursivos es $\Omega(2^n????????)$, mientras que la cantidad de subinstancias es $O(n^2)$. Por lo que presentamos una cantidad de llamadas exponencial mientras que las subinstancias son mucho menores al ser polinomial. Recordando que n es la cantidad de aspersores *válidos*[3] dentro del terreno.

Para la memoización primero optamos por el enfoque **Top-Down**. Empleamos como estructura para guardar los datos ya calculados una matriz de $n+1 \times n+1$. Este algoritmo tiene una *complejidad temporal* y *espacial* de $O(n^2)$ (por la dimensión de la matriz). Su pseudocódigo es el siguiente:

Algorithm 2 menorCostoTopDown(A):

```
Se inicializa M de n + 1 × n + 1 con
return f(0,-1) tal que
f(i,j):
if estaLleno(j) then
  return 0
end if
if noSeLlenara(i,j) then
  return -1
end if
if M[i,j] =  then
  if noSeLlenara(i+1,i) then
  M[i,j] ← f(i + 1, j)
  end if
end if
i ← 0
cubierto ← 0
minAspersores ← 0
return minAspersores
```

Luego fuimos por un enfoque **Bottom-Up** para poder comparar ambos métodos y ver realmente cuál es el que ahorra más tiempo de ejecución al eliminar pasos redundantes en el árbol recursivo. Se volvió a disponer, en principio, de una matriz de $n+1 \times n+1$, pero en vez de tener un algoritmo **recursivo**, este es **iterativo**. Se itera a la matriz de *izquierda a derecha* y de *abajo*

[3]Llamamos a un aspersor **válido** si su diámetro de riego puede cubrir el ancho del terreno.

a arriba. Es decir, cuando estamos parados en una *fila i*, la recorremos desde su *primera* posición hasta su *iésima* posición. Esto se repite partiendo desde la *n+1-ésima fila* hasta la *primera*. Como en la funcion f anteriormente mencionada el parámetro i nunca es mas grande que el parámetro j, no se visitan aquellas celdas donde el número de la fila sea mayor al de la columna. Para finalizar con la explicación, el algoritmo queda con *complejidad temporal y espacial* $O(n^2)$ (por la matriz también). Teniendo esto en mente, mostramos el algoritmo:

PSEUDOCODIGO

Al finalizar la implentación **Bottom-Up** notamos algo clave. Se puede *ahorrar espacio* con la estructura de memoización. No hace falta tener una matriz de $n+1 \times n+1$ para representar todas las soluciones, (tanto parciales como candidatas). Al no tener que reconstruir la solución ya que sólo se debe devolver el coste y no los aspersores que pertenecen al conjunto óptimo, podemos **desechar la matriz** y tener como estructura de memoización un **vector** de longitud $n+1$, guardando solamente la *fila anteriormente calculada* y no todas las filas anteriores como sucedía con la matriz. Esto no mejora la *complejidad temporal* pero sí la *espacial*, dejándola en $O(n)$ en vez de $O(n^2)$. Una mejora considerable en utilización de memoria. Su implementación quedó de la siguiente forma:

PSEUDOCODIGO

COMPARACION DE TIEMPOS DE EJECUCION ENTRE TOP-DOWN Y BOTTOM-UP PARA ELEGIR A UN GANADOR.

CONCLUSIÓN

Referencias

[1] https://www.blablabla.com/blabla/

[2] https://www.blablabla.com/blabla/

[3] https://www.blablabla.com/blabla/