DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Técnicas Algorítmicas

23 de marzo de 2023 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Embon Eitan	610/20	eembon1@gmail.com
Fiorino Santiago	516/20	fiorinosanti@gmail.com
Halperin Matias	1251/21	matias.halperin@gmail.com
Valentini Nicolas	86/21	nicolasvalentini@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - Pabellón I
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar

1. Backtracking

1.1. Robots on Ice

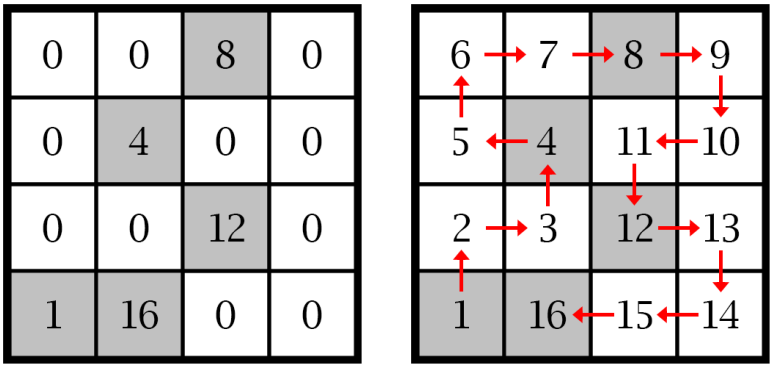
En este problema debemos determinar la cantidad de rutas posibles que tiene un robot de recorrer una grilla rectangular. El robot puede moverse sólo una celda por vez, en dirección Norte, Sur, Este u Oeste. La ruta debe visitar cada celda de la grilla exactamente una vez. Además, hay tres posiciones de "check-in" que el robot debe visitar cuando se encuentre a un cuarto, la mitad, y tres cuartos de su ruta completa. El robot debe empezar en la posición $(0, 0)$ de la grilla, y terminar en la posición $(0, 1)$.

1.2. Representación

La grilla la representaremos con una matriz de $m \times n$ enteros, la cual llamaremos M. En cada celda guardaremos el paso en el que el robot visitó esa posición. Inicialmente todas las celdas comienzan con valor 0, indicando que todavía no fueron visitadas, exceptuando los siguientes casos:

1. La posición $(0, 0)$ comienza con valor 1, ya que ahí es donde debe estar en el primer paso.
2. Las tres posiciones de los "check-ins" serán inicializadas con sus valores correspondientes (un cuarto, la mitad, y tres cuartos de la cantidad total de celdas).
3. La posición $(0, 1)$ comienza con valor $m \times n$ ya que el robot debe terminar en esa celda. Puede ser pensado como un cuarto "check-in".

En una solución válida esos 5 valores valores deben permanecer intactos. Además, no debe quedar ninguna celda con valor 0, y cada celda debe tener a su antecesor y sucesor en celdas adyacentes.



(a) Matriz inicial (b) Solución válida

Figura 1: Ejemplo con $m = n = 4$ y "check-ins" en $(2, 1)$, $(3, 2)$ y $(1, 2)$

1.3. Solución

Las soluciones válidas se van a construir efectuando un total de $m \times n$ pasos. Al momento de efectuar el paso s, la celda a la cual te trasladas cambia su valor a s.

La recursión general consiste en moverse hacia las 4 posiciones adyacentes. Si en el paso s del camino el robot se encuentra en la posición (i, j), consideramos las posiciones $\{(i-1, j), (i+1, j), (i, j-1), (i, j+1)\}$ que estén en rango de la matriz. Si bien esa recursión alcanza para encontrar todas las soluciones válidas, es ineficiente. Hay ciertos casos en los que podemos asegurar que la solución parcial no se puede extender a una solución válida, que son los siguientes:

1. Si en el paso $s + 1$ no debemos alcanzar ningún "check-in", la posición a la cual nos movemos debe tener valor 0.
2. Si en el paso $s+1$ debemos alcanzar el "check-in" de valor c, entonces la posición a la cual nos movemos debe tener valor c.

Con la condición 1. nos aseguramos que las celdas no se visiten más de una vez. Con la condición 2. nos aseguramos que siempre se llegue a los "check-ins" a tiempo, y que nunca se llegue a una posición que no sea un "check-in" en un paso en el que deberíamos haber alcanzado uno.

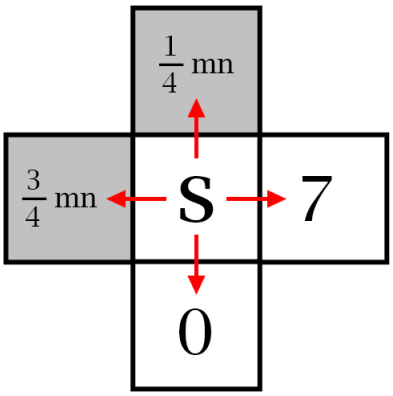


Figura 2: Posibles movimientos

En el ejemplo de la Figura 2 consideramos movernos hacia las 4 posiciones adyacentes. Siguiendo las condiciones explicadas anteriormente, tenemos 5 escenarios posibles:

- $s + 1 = (1/4)mn$
  Solo considera moverse para arriba.
- $s + 1 = (1/2)mn$
  No se puede extender a una solución válida, ya que en el paso $s + 1$ debemos visitar la celda con valor $(1/2)mn$, y no es ninguna de las adyacentes.
- $s + 1 = (3/4)mn$
  Solo considera moverse para la izquierda.
- $s + 1 = mn$
  No se puede extender a una solución válida, ya que en el paso $s + 1$ debemos visitar la celda con valor mn, y no es ninguna de las adyacentes.
- *caso contrario*
  Solo considera moverse para abajo, ya que en este paso no se debe alcanzar ningún checkpoint y la celda no tiene que estar visitada.

Si bien estas podas al árbol de recursión mejoran mucho la eficiencia del algoritmo, no son suficientes. Para mejorar el rendimiento aún más, consideramos agregar ciertas optimizaciones.

1.4. Optimizaciones

1.4.1. Distancia Manhattan

La distancia Manhattan nos indica la cantidad de pasos mínimos que debemos efectuar para viajar desde un punto (x_1, y_1) hasta otro punto (x_2, y_2). Esta distancia está definida como:

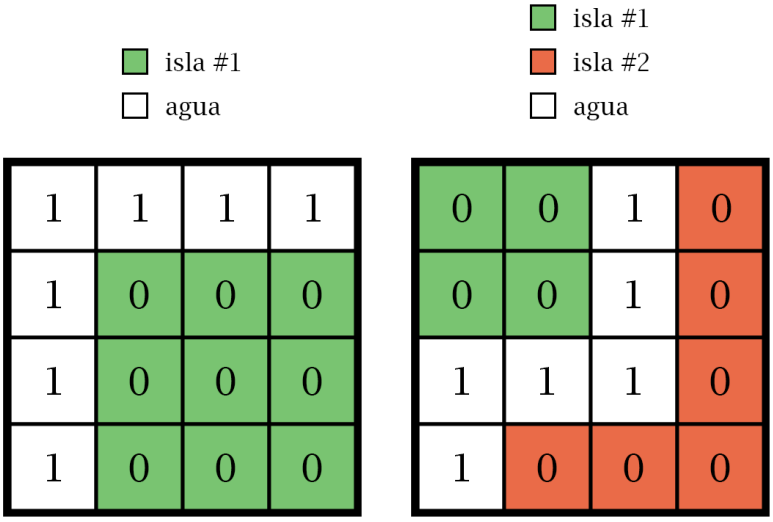
$$d = |x_1 - x_2| + |y_1 - y_2| \tag{1}$$

En cada paso de la recursión existe un "check-in" al cual debemos llegar en una cantidad determinada de pasos, ya que la posición $(0, 1)$ la tratamos como un último "check-in".
Sea s_{obj} el paso en el que debemos alcanzar el próximo "check-in", d la distancia Manhattan entre la posición actual y la posición del "check-in", y s_{curr} el paso actual. Definimos s_{left} como $s_{obj} - s_{curr}$, la cantidad de pasos restantes hasta el paso en el que debemos alcanzar el "check-in". Entonces existen dos opciones:

1. $d > s_{left}$
   Esto significa que necesitamos al menos d pasos para llegar al "check-in", sin embargo s_{left}, la cantidad de pasos restantes es menor, por lo tanto no hay forma de llegar a dicho "check-in" a tiempo. Esta solución parcial no se puede extender a una solución válida.
2. $d \leq s_{left}$
   Esto indica que todavía nos queda una cantidad de pasos mayor o igual a la distancia al próximo "check-in", por lo tanto todavía puede haber rutas válidas en las que alcanzamos dicho "check-in" en tiempo.

1.4.2. Problema de las Islas

Para la última optimización definimos el siguiente subproblema, que luego explicaremos cómo se usa en el contexto de nuestro problema original:
En el problema de las Islas se tiene una matriz binaria, la cual representa un mapa. Los ceros representan tierra, mientras que los unos agua. Una isla consiste en un grupo de una o más posiciones adyacentes que contienen tierra. El objetivo del problema es contar cuántas islas contiene una matriz.



(a) Matriz con una isla (b) Matriz con dos islas

Figura 3: Ejemplos del problema de las Islas

Como se puede observar en la Figura 3, en el primer caso todas las celdas con valor 0 están conectadas mediante adyacencias, por lo tanto tenemos 1 sola isla. En cambio, en el segundo caso tenemos dos bloques separados por agua.

La solución a este problema se obtiene haciendo un barrido lineal por la matriz, y en el caso de encontrar una celda con valor 0, se incrementa en 1 el contador de islas, y se aplica un DFS que en cada paso visita todas las celdas adyacentes de valor 0 y las marca con un 1. En otras palabras, cuando encuentra una isla la marca toda como agua, así cuando continúa con el barrido lineal no la cuenta más de una vez.

Este subproblema nos ayuda a nuestro problema original ya que podemos marcar las posiciones visitadas con un 1, y las no visitadas con un 0. Si a dicha matriz le contamos la cantidad de islas, podemos afirmar que si nos da un número mayor a 1, la solución parcial no se puede extender a una solución válida. Esto se debe a que sería imposible visitar las dos islas sin tener que cruzar el agua nuevamente. Para visitar las dos islas tendríamos que cruzar el camino ya recorrido, pero una de las condiciones para la validez de una solución es que cada celda se visita exactamente 1 vez. La solución parcial no se puede extender a una solución válida.
En el ejemplo de la Figura 3, en el primer caso tenemos una sola isla, lo cual implica que todavía podríamos extender dicha solución parcial a una solución válida (no necesariamente implica que se pueda). En el segundo caso, la cantidad de islas sube a dos, entonces podemos afirmar que no hay forma de extender dicha solución parcial a una solución válida. No importa cuál de las dos islas visites, te quedarías encerrado en ella.

2. Algoritmo Greedy

2.1. Watering Grass

En este ejercicio debemos resolver el siguiente problema: Dados n aspersores instalados en un terreno rectangular de l metros de largo y w metros de ancho, hallar el número mínimo de aspersores que debemos encender para cubrir el área completa del rectángulo. Cada aspersor está instalado sobre la línea horizontal central del rectángulo.

2.2. Instancias del problema

Como instancias del problema se reciben como parámetros w, $n \leq 10000$, l y un conjunto $A = \{a_1, a_2, ..., a_n\}$, siendo $a_i = (p_i, r_i)$ una tupla de dos enteros. A representa el conjunto de aspersores. Cada aspersor es representado como un circulo colocado a lo largo del rectángulo. Como la altura de su centro esta fijada (en la mitad del rectángulo), nos interesan tan solo dos parámetros para describirlos, la posición a lo largo del eje horizontal p_i y su radio r_i. Si fijamos la línea horizontal central del rectángulo como el eje x de coordenadas, entonces el área cubierta por un aspersor podrías ser descripta como:

$$area_i : \{(x, y) \ / \ (x - p_i)^2 + y^2 \leq r_i^2\} \tag{2}$$

2.3. Estrategias Posibles

En esta sección queremos encontrar la mejor estrategia golosa para resolver este problema. ¿Como sería una estrategia golosa? En un primer paso, podríamos empezar eligiendo el aspersor que tenga el radio mas grande desde un extremo del rectángulo. Luego podríamos cortar el rectángulo en el punto donde termina el aspersor elegido, y repetir el proceso en el área sobrante.

Una posible idea entonces, podría ser redefinir el conjunto A como $a_i = (p_i - r_i, p_i + r_i)$ y luego lo ordenamos de forma creciente respecto a su primera componente. De esta forma nos aseguramos que encontrar el aspersor de mayor radio sería ir iterando en el conjunto A y quedándonos con el candidato a_i que maximice $p_i + r_i$ pero que cumpla que $p_i - r_i$ es menor o igual que el comienzo del rectángulo a cubrir. Para un mejor entendimiento, una posible implementación podría ser la siguiente:

Sea $\#MinAspersores = 0$
Sea $start = 0$ y $best\_end = start$ // La variable start en esta instancia es el comienzo del rectángulo
while ($i < n$ && $p_i - r_i <= start$)
    $best\_end = max(r_i + p_i, best\_end)$;
    i++;

Si $start < best\_end$ entonces sumamos uno en $\#MinAspersores$, fijamos $start = best\_end$ y realizamos el ciclo nuevamente.

Problema

El problema de esta estrategia es que se está calculando mal el área que cubre un aspersor. Si definimos el intervalo que cubre como $I = (p_i - r_i, p_i + r_i)$, en los bordes nos podría quedar área sin cubrir. Esto llevaría a que el algoritmo determine que hay solución cuando en realidad no existe.

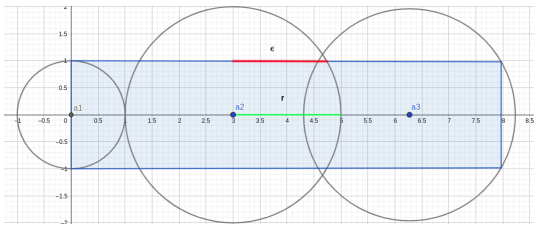


Figura 4: Ejemplo de una instancia del problema, donde $|A| = 3$. El segmento **r** representa el radio de a_2 y luego el segmento ϵ es una medida alternativa para reemplazar la función de **r**.

Solución

Una posible solución a este problema es tomar la distancia ϵ, que es la distancia horizontal desde p_i hasta el extremo del circulo, pero ahora en vez de medirla sobre la mitad del rectángulo (en ese caso sería el radio), la medimos sobre $y = w/2$ (el borde superior del rectángulo). De esta forma nos estamos asegurando que cuando *cortamos el rectángulo* en *best_end*, el área anterior a ese punto esta cubierta.

En este punto tenemos que definir como obtenemos ϵ. Observemos que este parámetro es en realidad la mitad del segmento intersección entre la recta $y = w/2$ (la parte superior del rectángulo) y la ecuación del área que cubre un aspersor descrita en 2. Nos queda por resolver el siguiente sistema:

$$\begin{cases} (x - p_i)^2 + y^2 \leq r_i^2 \\ y = w/2 \end{cases}$$

Reemplazando y despejando x obtenemos que

$$p_i - \sqrt{r_i^2 - (w/2)^2} \leq x \leq p_i + \sqrt{r_i^2 - (w/2)^2}$$

Y con esto obtenemos que $\epsilon_i = \sqrt{r_i^2 - (w/2)^2}$.

2.4. Solución final

Utilizando la última estrategia, podemos redefinir el problema como: Sea un aspersor definido $a_i = (a_i^{(1)}, a_i^{(2)}) = (p_i - \epsilon_i, p_i + \epsilon_i)$ con $p_i \in \mathbb{Z}, \epsilon_i \in \mathbb{N}$. Dado un conjunto de aspersores $A = \{a_1, \ldots, a_n\}$ y $l \in \mathbb{N}$. Debemos encontrar una secuencia $S = s_1 \ldots s_k$ tal que $S \in \mathcal{S}$ que minimice k con:

$$\mathcal{S} : S \subseteq A \; tal \; que \begin{cases} s_i^{(1)} \leq s_{i-1}^{(2)} \leq s_i^{(2)}, & i = 2 \ldots k \\ s_1^{(1)} \leq 0 \leq s_1^{(2)} \\ s_k^{(2)} \geq l \end{cases} \tag{3}$$

El óptimo quedaría definido como:

$$\Pi = \min_{S \in \mathcal{S}}(|S|) \tag{4}$$

Para resolver este problema podemos intentar generar una solución $S = s_1 \ldots s_k$. Una forma simple de generar $S \in \mathcal{S}$ es en forma iterativa: en el i-ésimo paso buscamos s_i pero antes fijado un *pivote*. Usando la definición de S en la ecuación 3 buscamos un a_l candidato a s_i tal que maximiza $a_i^{(2)}$ cumpliendo además que $a_i^{(1)} \leq pivote \leq a_i^{(2)}$, luego actualizamos $pivote = a_i^{(2)}$. Si $pivote \geq l$ retornamos $|S|$. Los valores posibles para nuestro pivote son $[0, a_1^{(2)}, a_2^{(2)}, \ldots, a_n^{(2)}]$. El algoritmo resultante de esta construcción es goloso porque en cada paso se toma la mejor elección local.

Podríamos seguir los siguientes pasos:

1. Reescribir A tal que $a_i = [p_i - \epsilon_i, p_i + \epsilon_i]$
2. Ordenar A por la primer componente, y en caso de empate por la segunda.
3. Fijamos un comienzo en 0 y cantidad de aspersores necesitados en 0.
4. Buscamos a_i que maximice $p_i + \epsilon$ tal que $p_i - \epsilon_i \leq$ comienzo y sumamos uno a la cantidad de aspersores necesitados.
5. Reemplazamos comienzo $= p_i + \epsilon$ y realizamos una nueva búsqueda. Seguimos iterando hasta que comienzo sea igual al final del rectángulo.
6. Retornamos la cantidad de aspersores necesitados, y si estos son cero, retornamos -1.

2.5. Complejidad

Temporal

Los ítems 3. y 6. de la solución descripta son asignaciones de enteros a variables por lo que consideramos $O(1)$. El ítem 1. es reescribir A, con lo que nos toma $O(n)$. El ítem 2. ordena un conjunto de tamaño n, con lo que la complejidad podriamos tomarla como $O(nlogn)$. Por último en los ítems 4. y 5. realizamos una búsqueda en A tantas veces hasta que comienzo = l. Podría pasar que los aspersores estén ubicados de tal manera que recién para a_n pase que $p_n + \epsilon_n \geq l$, con lo cual nos quedaría con n repeticiones de una búsqueda en un conjunto de tamaño n. Por lo tanto, decimos que los ítmes 4. y 5. tienen complejidad $O(n^2)$. Sumando las complejidades mencionadas, nos queda que el algoritmo especificado tiene *complejidad temporal* $O(n^2)$.

Espacial

En todo el algoritmo se utiliza el conjunto A como único elemento guardado en memoria (acotando a los tipos simples, ejemplo asignaciones de enteros), lo que nos da un tamaño $O(n)$. A esto le debemos sumar la complejidad espacial del método de ordenamiento a usar $O(sort)$.

Esto nos deja con una complejidad espacial $O(\max(sort, n))$

2.6. Demostración

Sea S una solución óptima del problema, veamos que la solución G que encuentra el algoritmo goloso tiene una cantidad menor o igual de aspersores (es óptima).

Si $S = G$ es trivial, tiene la misma cantidad de aspersores. Ahora veamos el caso en el que la solución óptima difiere de la golosa. Siendo $S \neq G$, existen tres opciones:

1. $|S| > |G|$: Absurdo. Si la solución óptima tiene más aspersores que la solución golosa, entonces no es óptima.
2. $|S| = |G|$: La solución golosa tiene la misma cantidad de aspersores que la solución óptima.
3. $|S| < |G|$: La solución golosa no es óptima.

Teniendo en cuenta que el primer caso es absurdo y en el segundo la solución golosa es óptima, para demostrar que la solución golosa *siempre* es óptima, alcanza con demostrar que el tercer caso es imposible que suceda.

Supongamos que las soluciones tienen sus elementos ordenados por su primera componente, es decir, el comienzo del intervalo del terreno que cubre cada aspersor. También supongamos que $s_i = g_i$ $(\forall i = 1, ..., k-1)$, y las soluciones difieren en el aspersor k.

Teorema: Sean $S, G \subseteq A$, $|S| = |G| = k$ dos conjuntos de aspersores ordenados por su primera componente. Si cumplen la condición $s_{i+1}^{(1)} \leq s_i^{(2)}$ (los aspersores están enganchados). Si además $s_k^{(2)} \leq g_k^{(2)}$ (el área que cubre G es mayor o igual que el área que cubre S). Entonces si a G le agregamos el aspersor $a \in A$ tal que $a^{(1)} \leq g_k^{(2)}$ (está enganchado a G) con mayor segunda componente, podemos afirmar que agregándole a S cualquier $a \in A$ tal que $a^{(1)} \leq s_k^{(2)}$ (está enganchado a S), la solución G va a seguir cubriendo un área mayor o igual a la que cubre S.

Demostración: Sabemos que $s_k^{(2)} \leq g_k^{(2)}$. Esto implica que el subconjunto de aspersores que pueden ser agregados a la solución S va a tener menor o igual cantidad de elementos que el subconjunto de aspersores que pueden ser agregados a la solución G. Este subconjunto será el mismo en caso de que no haya ningún aspersor que tenga su primera componente entre $s_k^{(2)}$ y $g_k^{(2)}$, en caso contrario G considerará más aspersores que S. Además, los aspersores que considera S son un subconjunto de los que considera G. Por lo tanto, el aspersor que tomará G será uno de mayor o igual segunda componente que el que considerará S, cubriendo así mayor área.

Volviendo a la demostración original, las soluciones coinciden hasta el aspersor $k-1$. Ahora, para el aspersor k, la solución golosa tomó el aspersor enganchado con mayor componente secundaria, por lo tanto la solución óptima tomó uno con menor o igual componente secundaria que la golosa. Por el momento ambas soluciones tienen la misma cantidad de aspersores, los cuales están ordenados por primera componente.

Además, G cubre un área mayor o igual a la que cubre S, por lo tanto, se cumplen todas las hipótesis necesarias para el teorema enunciado anteriormente. Usando el teorema, podemos afirmar que efectuando un paso más del algoritmo goloso, y agregando otro aspersor enganchado a S, entonces G va a seguir cubriendo un área mayor o igual a la que cubre S. Como se sigue cumpliendo la hipótesis, podemos repetir este proceso hasta cubrir todo el rectángulo. Como en cada paso la solución óptima cubrirá un área menor o igual a la golosa, nunca podrá llegar a cubrir todo el rectángulo en una menor cantidad de pasos, es decir, agregando una menor cantidad de aspersores. Por lo tanto, nunca se puede dar el caso de que $|S| < |G|$.

Una vez demostrado que este caso no se puede dar, queda demostrado que la solución G es óptima, ya que éste era el único caso en el que G podía no ser una solución óptima. Entonces, siempre que exista solución, el algoritmo goloso encontrará una solución óptima.

3. Programación Dinámica

3.1. Implicancias del agregado de costos

Al agregar un costo a cada aspersor, la solución greedy deja de funcionar, y se convierte en un problema combinatorio. Se puede generar cierto paralelismo con el problema de la mochila, pero en ese caso el peso que soporta la mochila es un entero, por lo que se puede discretizar. En el caso del césped, hay infinitos puntos en el intervalo $[0, l]$. Lo que pudimos observar, sin embargo, es que no todos esos puntos son posibles intervalos $[0, p]$ a cubrir por los aspersores. En su lugar, si tomamos el final de cobertura de cada aspersor, entonces en el mejor de los casos se podrían cubrir los intervalos $[0, e]$ siendo e todos los finales de cobertura de los aspersores. Estos son, por ser n aspersores, a los sumo n puntos. Como logramos discretizar nuestro problema en n aspersores y n posibles áreas de cobertura, entonces sabemos que con una matriz de $n \times n$ podemos almacenar todas las subinstancias. Esto nos da una idea de una cota n^2 para la programación dinámica en caso de lograr resolver cada una en $O(1)$.

3.2. Precalculo del "enganche" de cada aspersor

Para cada aspersor, hay un inicio de cobertura y un fin de cobertura. El fin de cobertura es el que usamos para discretizar el césped, y al que vamos a referirnos siempre como p. El inicio, por otro lado, es el que nos sirve para evaluar si dos intervalos "enganchan", es decir, si su unión forma un intervalo continuo. Vamos a llamarlo s. Es por esto que, para cada p, nos interesa conocer el mínimo s tal que s $\leq$ p. Averiguar esto dentro de la recursión nos tomaría más que $O(1)$, es por eso que decidimos precalcularlo.

3.3. Caso recursivo

En cada caso recursivo vamos a tener un k, que representa a los k primeros aspersores con los que se cuenta, y un p, que representa el punto hasta el que hay que regar (el área cubierta por el regado debe contener al intervalo $[0, p]$). De los k aspersores que tenemos para cubrir, el último es el que se acaba de agregar, por lo que tenemos que decidir si usarlo o no. Nos encontramos entonces con 3 opciones:

1. El inicio de la cobertura es menor o igual a 0, y el final mayor o igual a p. En este caso, el aspersor en si mismo puede cubrir todo el rango $[0, p]$, por lo que lo óptimo es elegir entre 2 opciones:
   - El costo de cubrir este área sin usar el aspersor
   - El costo del aspersor
2. El área de cobertura del aspersor se encuentra totalmente adentro del intervalo $[0, p]$, o totalmente afuera. En este caso, no nos sirve el aspersor, por lo que lo descartamos.
3. La cobertura de éste inicia en un punto menor o igual a p, por lo que decimos que el aspersor "engancha" con el intervalo. En este caso, el costo óptimo es el mínimo de:
   - Cubrir el intervalo $[0, p]$ sin el aspersor
   - El costo de cubrir el intervalo $[0, s]$ más el costo de este aspersor. Usamos el mejor costo de $[0, s]$ en lugar de $[0, p]$ ya que podría haber aspersores cuyo inicio y fin se encuentre en el rango $[s, p]$. En este caso, prender esos aspersores sería sumar un costo innecesario, ya que ese rango se encuentra cubierto por el aspersor.

3.4. Función recursiva

Notación: f recibe un i, el i-ésimo aspersor (lo que implica el uso de los primeros i aspersores), y un p, que representa el intervalo $[0, p]$ a cubrir. s y e son el inicio y fin de la cobertura del i-ésimo aspersor, respectivamente. c es el costo del i-ésimo aspersor. *precalc* es el arreglo precalculado explicado anteriormente.

Cuando no hay ningún aspersor con $e \geq l$, no hay ningún punto que resuelve el problema. En este caso la respuesta es inmediatamente -1. En cualquier otro caso, se llama a la función con $i = n$ y siendo p el mínimo e tal que $e \geq l$.

La función recursiva quedaría dada entonces de la siguiente forma:

$$f(i,p)\begin{cases}\infty, & \text{si } i=-1\\ \min(f(i-1,p),c), & \text{si } s\leq 0 \text{ y } e\geq p\\ f(i-1,p), & \text{si } s>p \text{ ó } e\leq p\\ \min(f(i-1,p),f(i-1,precalc[i])+c), & cc\end{cases}$$

3.5. Superposición de subproblemas

En el peor caso, todos los casos recursivos llaman a otros dos (el caso cc). Esto nos da $\Omega(2^n)$ llamadas. Como los parámetros i y p están en el rango $[0,n)$, tenemos $O(n^2)$ subinstancias. Como $O(n^2) << \Omega(2^n)$, entonces nos conviene memoizar las soluciones en una tabla de n × n. Esto nos permitiría resolver la recursión en $O(\#subinstancias \times costo\_subinstancia)$.

3.6. Complejidad

El precalculo consiste en ordenar los aspersores en base a e ($O(n^2)$) y luego, por cada aspersor i, iterar hasta encontrar al primero cuyo e sea mayor que el s de i (también $O(n^2)$ por iterar n veces en cada uno de los n aspersores). Encontrar al mínimo e $\geq$ l cuesta $O(n)$. Por último, la recursión con programación dinámica resuelve a lo sumo n^2 subinstancias, cada una en $O(1)$ gracias al precalculo. Esto implica que todo el problema tiene una cota de complejidad $O(n^2)$.