# Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

## Trabajo Práctico 2



| Integrante | LU | Correo electrónico |
|---|---|---|
| Ortiz de Zarate, Juan Manuel | 403/10 | jmanuoz@gmail.com |
| Martelletti, Pablo | 849/11 | pmartelletti@gmail.com |
| Kujawski, Kevin | 459/10 | kevinkuja@gmail.com |
| Carreiro, Martin | 45/10 | carreiromartin@gmail.com |



**Reservado para la cátedra**

| Instancia | Docente | Nota |
|---|---|---|
| Primera entrega | | |
| Segunda entrega | | |

# 1. Problema 1

## 1.1. Enunciado

El enunciado nos plantea una situación en donde tenemos un edificio con n pisos y personas en cada piso que quiere ir a planta baja. Para poder hacerlo, el edificio provee un ascensor que deberá buscar a las personas para poder bajarlas. Este ascensor posee energía y capacidad limitada que no le permite recorrer siempre todos los pisos y levantar todas las personas, por lo tanto queremos maximizar la cantidad de personas a descender del edificio dado la cantidad de personas por piso, su energía y capacidad.

## 1.2. Solución

La solución planteada utiliza Programación Dinámica a través de decisiones. Planteamos el problema de forma tal que en vez de maximizar la cantidad de personas que pueden descender, encontramos el máximo valor posible que va a estar ubicado entre cero y la cantidad de personas totales en el edificio.
A continuación se explicará cómo se resolvió el problema:
Primero obtendremos la cantidad total de personas recorriendo el edificio dado, y buscaremos el máximo valor de personas que podremos llegar a descender utilizando la búsqueda binaria entre la cantidad de personas en el edificio, hasta encontrar el valor más alto que se pueda levantar, tal que incrementado en uno no sea posible levantarlo. En cada paso de la búsqueda binaria, hay que comprobar si se puede levantar cierta cantidad de personas (dada la capacidad del ascensor, su energía y la cantidad de pisos del edificio). Para comprobar que se puedan levantar buscaremos el piso mínimo al que necesitaremos ir*. Recorreremos el edificio desde el piso mínimo hacia abajo solo en caso de ser posible llegar hasta él y poder volver a descender, levantando la máxima cantidad de personas de cada piso en el camino hasta llenar la capacidad total del ascensor. Una vez completo el ascensor, descenderemos a los pasajeros hasta planta baja y volveremos a repetir el proceso. En el caso de que haya quedado gente y nos quede energía volveremos al mismo piso, en caso contrario seguiremos con el siguiente piso al que podamos ir hasta que levantemos la cantidad requerida o hasta que nos quedemos sin energía (caso que devolveremos que no se puede levantar esa cantidad de gentar). Tener en cuenta que el piso mínimo elegido puede dejarnos en la parte inferior del edificio con más personas de las requeridas. Entonces al momento de levantar puede pasar que no levantemos gente y que sin embargo, descendamos la cantidad requerida de personas.
Para la demostración de la resolución explicaremos las razones por las cuales el piso mínimo elegido previamente es el correcto y la estrategia de levantar de arriba hacia abajo es verdadera. Este piso mínimo es la mejor opción ya que si subimos más pisos estaríamos utilizando energía para levantar la misma cantidad de gente que podríamos encontrar en los pisos inferiores, y, no se puede ir más abajo porque si no, no tendríamos la cantidad de gente necesaria para poder levantar el valor pedido.
Definimos el mundo de las estrategias como el conjunto de estrategias posibles y estrategia como el método por el cual se levanta gente. Como no nos interesa la energía utilizada, ya que solo queremos ver si puede levantar la cantidad de personas mencionada, y no la optimalidad de energía, una estrategia estará conformada por la cantidad de personas que el ascensor levanta en cada piso. De este mundo de estrategias tomaremos solo aquellas en las que su piso más alto del cual levantan personas es el nuestro, y que levanten la misma cantidad que nosotros. Este conjunto es distinto de vacío ya que es un conjunto acotado de personas y pisos, por lo tanto se van a poder levantar a partir de cierta energía. Ahora queremos ver que nuestra estrategia se encuentra en este conjunto para lo que tomaremos una de esas estrategias E y veremos que se pueda permutar de tal forma que consigamos nuestra estrategia. Vamos a dividir a las estrategias verdaderas entre las que levantan mientras suben y el resto.
El caso de que E levante mientras sube, ya sabemos que el piso máximo de E es el mismo que el nuestro, que podemos ir hasta el piso máximo y al bajar vamos a pasar por los mismos pisos por lo que se terminan levantando la misma cantidad de personas. Puede llenarse antes en los pisos superiores, pero siempre vamos a poder llenar con los pisos que E llenó.
Para el caso desordenado, podremos ubicar la forma que recorre el edificio tal que levante de forma ascendente. Esto sucede ya que si el edificio pasa por los pisos 1 hasta el máximo pero en el medio va agarrando personas en forma desordenada, podemos permutar el método y que vaya levantando mientras asciende.
Por lo tanto, tenemos que el segundo caso puede llevarse al primero, y este último hacia el nuestro; y como las estrategias que tomamos son las que funcionaban, entonces la nuestra también funciona.

***** Para ubicar este piso mínimo recorreremos el edificio desde abajo contando la cantidad de personas en los pisos, hasta que el valor sea mayor o igual al requerido.

## 1.3. Pseudocódigo

```
RESOLVER (in capacidad, in energia, in pisos) ⟶ maximaCantidad : Int
1  totalPersonas = SumaTotalPersonas(pisos);
2  return BusquedaBinariaPersonas(0,totalPersonas, totalPersonas,energia,capacidad,pisos);
```

---

```
SUMATOTALPERSONAS (in pisos) ⟶ acum : Int
1  Para cada piso p
2        acum = acum + genteEnPiso(p)
```

---

BusquedaBinariaPersonas (**in** *desde*, **in** *hasta*, **in** *total*,**in** *energia*, **in** *capacidad*, **in** *pisos*) $\longrightarrow$ *acum* : Int

```
1  medio = (desde + hasta)/2;
2  puedo = sePuedeLevantar(energia, capacidad, pisos, medio);
3  Si puedo
4        Si no se puede levantar el siguiente terminé
5        Si no busquedaBinariaPersonas en la siguiente mitad
6  Si no busquedaBinariaPersonas en la primera mitad
```

---

sePuedeLevantar (**in** *energia*, **in** *capacidad*, **in** *pisos*, **in** *personasABuscar*) $\longrightarrow$ *sePuede* : Boolean

```
 1  Para cada piso empezando desde el piso minimo
 2        Si tengoEnergiaParaLlegar y Hay Gente
 3              Si el piso tiene mas gente que mi capacidad
 4                    sumoLoQueLevante
 5                    veoSiPuedoVolverAEstePiso
 6              Si no
 7                    sumoLoQueLevante
 8                    mientrasBajoSumoGenteParandoEnLosPisosQueHayGente
 9        Si no
10              AnotoLasQueNoLevanté
11  return (cuantoLevante ¿= personasABuscar)
```

---

## 1.4. Analisís de complejidad

Para averiguar cuanta gente hay en todo el edificio tenemos que recorrer todo el vector 'pisos' e ir sumando la cantidad de gente en cada piso. Eso nos cuesto O(n) donde n es la cantidad de pisos. Luego hacemos una búsqueda binaria sobre la cantidad total de gente que mediante la función sePuedeLevantar busca el máximo de gente que es posible cargar. La complejidad de la búsqueda binaria es de O(log (cantGente)) (esto es sabido, porque ya fue visto en la materia de Algoritmos y Estructura de Datos II) y la complejidad de la función sePuedeLevantar en el peor caso (sería cuando tiene que irse hasta el último piso para poder levantar la cantidad de gente solicitada y tiene energía suficiente para levantarlos a todos) es de:

$O( \sum_{i=0}^{n} (\lceil (pisos[i]/capacidad) \rceil + i \ ) \ )$

Esto es porque por cada piso al que voy tengo que ir tantas veces tal que levante el total de gente de ese piso (eso es ⌈pisos[i]/capacidad⌉) y luego en caso que me haya sobrado espacio voy recorriendo todos los pisos inferiores para llenar la capacidad restante. Finalmente la complejidad total del ejercicio nos queda:

$O( \ n + Log(g) * \sum_{i=0}^{n} (\lceil (pisos[i]/capacidad) \rceil + i) \ )$

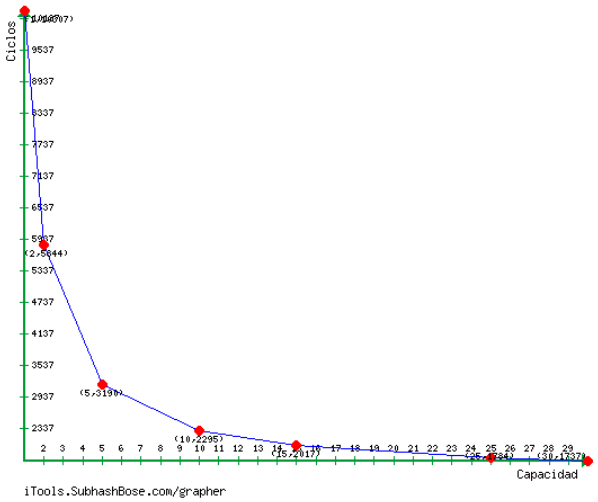
Donde n es la cantidad total de pisos y g la cantidad total de gente en todo el edificio.

## 1.5. Tests y Gráficos

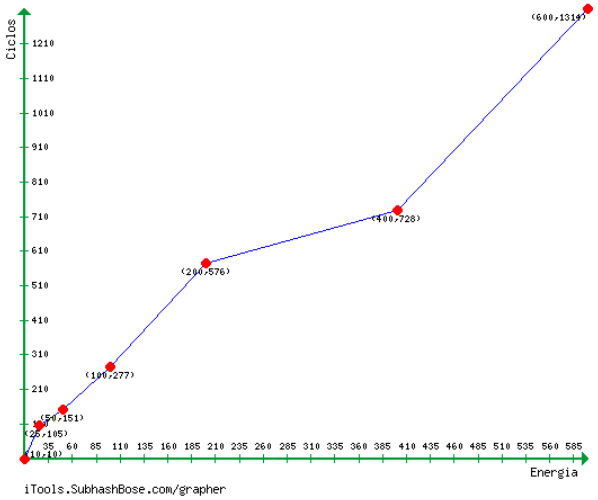
En los test quisimos demostrar que cuanto menos capacidad tenga el ascensor (siempre y cuando tenga energía suficiente) mas ciclos va a requerir la resolución. Y por otro lado que la energía es un factor limitante ya que cuanto menos energía tenga el ascensor no solamente va a levantar menos gente sino que ademas va a ciclar menos. Para los dos gráficos utilizamos el mismo edificio (ya que para lo que queremos demostrar no necesitamos variar el edificio), que es el siguiente:

PISO 12 — 45 —
PISO 11 — 20 —
PISO 10 — 60 —
PISO 9 — 90 —
PISO 8 — 80 —
PISO 7 — 45 —
PISO 6 — 8 —
PISO 5 — 38 —
PISO 4 — 46 —
PISO 3 — 66 —
PISO 2 — 11 —
PISO 1 — 4 —

En el siguiente gráfico dejamos fija la energía en un valor muy grande (9700) para que la mimsa no sea un factor limitante.Aquí podemos observar como disminuye la cantidad de ciclos a medida que aumenta la capacidad. Esto es por el factor (pisos[i]/capacidad) que mencionamos en la complejidad, por trivialidad matemática ese factor aumenta a medida q disminuye la capacidad.




En el siguiente gráfico dejamos fija la capacidad en un valor chico (5) para demostrar como la energía limita la cantidad de ciclos. Decimos que limita ya que no sólo no nos restringe la cantidad de personas a levantar sino que ademas descartamos muchos casos al resultar imposibles por esta cuestión energética.
Claramente podemos observar como los ciclos aumentan al tener mas energía. Notar que también aumenta porque hay mucha gente para levantar y poca capacidad, podría pasar que hubiese un solo piso con 1 persona y en caso, por ejemplo, la energía acotaría muy poco los ciclos, ya que se requieren pocos ciclos para levantar la máxima cantidad de gente.




## 1.6. Conclusiones

El problema nos mostró que podía ser encarado de distintas maneras dentro de la métodología de programación dinámica. Al principio habíamos utilizado una idea parecida a la de SubSetSum con memorization pero percances en el desarrollo nos hicieron repensar. Por esto, encaramos el problema a través de decisiones lo que nos ayudó a entender el principio de optimalidad.

# 2. Problema 2

## 2.1. Enunciado

El enunciado nos plantea una situación en donde tenemos $n$ localidades conectadas entre si, mediante uno o más enlaces telefónicos. La empresa desea que la conectividad entre todas las ciudades nunca se pierda, por lo que está interesada en invertir en cableado subterraneo indestructible. El precio de conectar las distintas localidades varía según las mismas, por lo que la empresa nos pide que hallemos la forma de conectar todas las ciudades y que ninguna quede incomunicada, de forma de que la inversión en éste nuevo cableado subterraneo sea mínima.

## 2.2. Solución

Al presentarnos éste problema, inmediatamente pensamos que era posible reprentarlo mediante un problema de grafos. El mismo representaría todo la red de conectividad de la empresa telefónica, los vértices del mismo representarían las localidades y las aristas entre los distintos nodos representarían el costo del cableado subterraneo entre las ciudades que uniera la misma. Dicho ésto, el problema se reducía, entonces, a hallar un grafo cuyas aristas permitan ir de una ciudad a cualquier otra del grafo, y además, a que esas aristas tengan un costo mínimo de construcción. En otras palabras, necesitamos un algoritmo que nos calcule, dado un grafo cualquiera, el arbol generador mínimo (AGM). El agm es un grafo que contiene los mismo vértices que el original, pero sólo contiene v-2 aristas (siendo v la cantidad de vértices), que unen todo el grafo y cuyo peso total es mínimo. Para el problema, hemos implementado una clase propia de Grafo, representanda por una lista de adyacencia, y utilizado el conocido y demostrado Algoritmo de Prim para generar el AGM. El mismo consiste en, dado un grafo cualquiera, agarrar un vértice del mismo al azar, y agregarlo al AGM (que originalmente no tenía ningún vértice ni aristas). Luego, agregamos al grafo la arista de menor peso que conecte al vértice recién insertado con un vértice que no esté en el agm. Para ello, agregamos todas las aristas del vértice en una cola de prioridad, y vamos desencolando hasta que encontremos una arista que tenga un extremo en el agm y otro no (es decir, tengo un sólo vértice en el árbol). Una vez que encontramos dicha arista, agregamos el vértice que no estaba al AGM y también la arista. El proceso debe ser repetido, de la misma forma, con todos los vértices del grafo, es decir, hasta que la cantidad de vértices que hay en el árbol sea la misma que había en el grafo original. Ésta es la forma en que se describe el algoritmo de prim y la forma en que actúa nuestro algoritmo, por lo que podemos decir que la implementación realizada se corresponde con dicho algorítmo.

## 2.3. Pseudocódigo

En código del ejercicio básicamente hace una cosa: lee el archivo de entrada, y procesa las lineas y va creando las distintas instancias del ejercicio. La lógica del problema está toda delegada en la clase Grafo, que es la que se encarga de modelar las localidades y sus enlaces (el grafo en sí), y luego, una vez modelado, de crear el propio AGM. El agm está implementado a partir del algoritmo de prim, que busca para cada vertice nuevo agregado, la arista de menor peso que lo conecte con un vertice que no esté en el grafo y la agrega. La lógica del ejercicio podría dividirse, de acuerdo a lo recién mencionado, de la siguiente manera:

```
CrearGRafo (String instancia)  ⟶  g : Grafo
1  Grafo g;
2  para cada enlace de la instancia:
3        agrego el vertice 1
4        agrego el vertice 2
5        agrego la arista entre v1 y v2
6  devuelvo el grafo
```

---

```
getAgm()  ⟶  g : Grafo
 1  Grafo agm;
 2  agrego un vertice al azar del grafo al agm
 3  marco el vertice agregado como visitado
 4  agrego las aristas del vertice visitado a la cola de prioridad
 5  para cada vertice en grafo:
 6        obtengo la arista de menor peso de la cola
 7        miro los dos vertices de la arista
 8        si alguno no fue visitado:
 9              agrego la arista minima al agm
10              marco el vertice como visitado
11              agrego las aristas del vertice a la cola de prioridad
12  devuelvo el agm
```

---

## 2.4. Analisís de complejidad

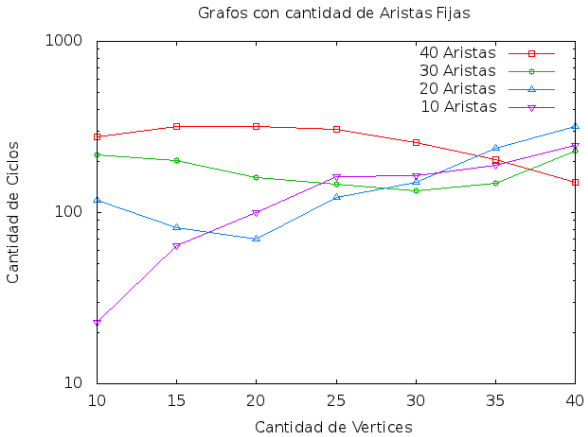
En éste ejercicio, para el análisis de la complejidad utilizaremos la información conocida de la documentación de Java, ya que en su gran mayoría, la complejidad de ejercicio depende de las estructuras que utilizamos para resolver el mismo. Para el análisis, sólo vamos a tener en cuenta lo que nos cuesta resolver el ejercicio, y no la forma en que leemos los archivos de entrada. Por ello, asumimos que leer de la entrada no afecta a la complejidad del algoritmo. Por su parte, y una vez que ya tenemos la entrada cargada en memoria, recrear las instancias va a depender de la cantidad de vértices que tengamos, y de las aristas que tenga cada vértice. Ingresar un vértice es $O(1)$ para cada uno de ellos, lo mismo que ingresar las n aristas que tenga dicho vértice. Por lo tanto, la complejidad de recrearlo será de $O(v+c) = O(v)$, siendo v la cantidad de vértices del grafo y c una constante, dependiente de las aristas que tenga cada vértice (el cual podemos obviar). Luego, para la resolucion del problema, recorremos 1 vez cada vértice, que tenemos almacenado en un arrayList. Obtener cada vértice es $O(1)$, por lo que recorrer los n vértices tendrá un costo de $O(n)$. Luego, para cada vértice, agregamos las aristas que lo tienen como uno de los extremos a una cola de prioridad. En el peor de los casos (por ejemplo, para el último vértice, donde ya ingresamos todas las aristas de todos los vértices), vamos a tener que ingresar e aristas (donde e es al número total de aristas del grafo). En ese caso, las acciones básicas de la cola de prioridad (ingresar un elemento, o quitar el primer elemento), tienen un costo de $O(\log e)$, por lo que realizarlo para todos los vértices, el costo total de las operaciones de la cola sería de $O(v \log e)$. Por útlimo, quedan operaciones básicas, como comparar valores, ver si el vértice está visitado o agregar el vértice al agm, que en todos los casos, son $O(1)$. Finalmente, la complejidad de nuestra solución será de

$$O(v) + O(v) + O(v \log e) = O(v + v \log e)$$

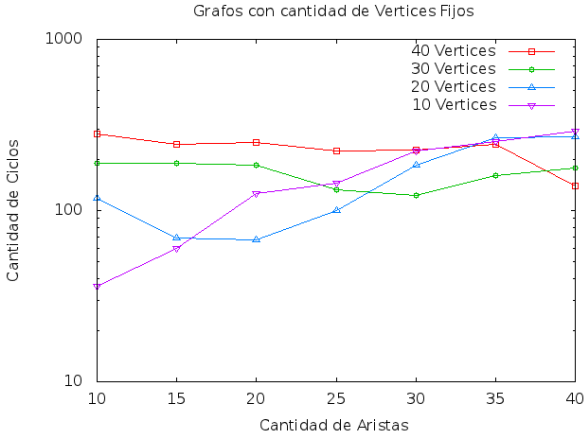
cumpliendo de ésta forma lo pedido en el enunciado, que establecía que el ejercicio debía resolverse en, como mucho, $O(m * v)$.

## 2.5. Tests y Gráficos

Con respecto a los tests realizados, los mismos no se hicieron en cuanto a la complejidad (ya que al utilizar estructuras primitivas de java, podemos asegurar que la complejidad de cada una de sus operaciones es la que aparece en la documentación de ellas y por tanto, es la que detallamos en el apartado anterior), sino en cuanto a la cantidad de ciclos que realiza cada solución (más precisamente, la creación del árbol generador mínimo) para distintas instancias, de acuerdo a la cantidad de vértices (localidades) y aristas (enlaces) que posee cada una. Como podemos apreciar en los siguientes gráficos, la cantidad de ciclos, para grafos en donde la cantidad de aristas es la misma, es proporcional a la cantidad de aristas totales que haya en el vértices, sin importar cuántos vértices haya o cuántas aristas por vértice tenga el grafo. Para mostrar ésto, realizamos dos gráficos, luego de correr varias pruebas de las siguientes instancias: por un lado, utilizamos como variable la cantidad de vértices, manteniendo las aristas por vértices constantes (figura 1). Y por otro lado, hemos variado la cantidad de vértics, manteniendo constantes la cantidad de aristas por vértice (figura 2). En ambos casos, la cantidad TOTAL de aristas del grafo se mantenía igual. Como podemos apreciar en primer gráfico, dónde hemos mantenido la cantidad de aristas por vértice constante, vemos que el comportamiento es el de esperar: a mayor cantidad de vértices, mayor cantidad de ciclos. En el mismo hemos corrido instancias con 10, 20, 30 y 40 aristas y hecho un promedio de los ciclos en 10 ejecuciones y, en su mayoría, se ha mantenido éste criterio. Es de esperar que, como pasa en el caso con más aristas, no siempre se cumpla que la cantidad de ciclos crece ya que, al tener que analizar muchas aristas, es posible que encuentre una arista de mayor peso en el primer paso, o bien en el último, siendo que los pesos de las aristas son creados de forma aleatoria.




Por otro lado, podemos apreciar el gráfico que hemos creado a partir de las instancias que poseen vértices fijos. Como vemos, es muy parecido al gráfico anterior, como era de esperar según nuestras suposiciones. Nuevamente, en el caso de mayor cantidad de vértices fijos, no siempre se cumple que a más aristas, más cantidad de ciclos, por la misma razón que en gráfico anterior. Pero, en promedio, podemos decir que a mayor cantidad de aristas totales en el grafo, más cantidad de ciclos, sin importar cómo estén distribuidas éstas: ya sea teniendo más aristas por vértice, o bien teniendo más vértices.




## 2.6. Conclusiones

A partir del problema dado, hemos podido modelar a partir de la teoría de grafos. De ésta forma, e interpretando qué es lo que nos pide el enunciado, hemos podido resolver el problema a partir de un algoritmo que es conocido y, de esa forma, podemos afirmar que estamos dando la solución correcta. En ésta caso, modelamos nuestras localidades y los distintos precios que costaban unirlas a partir de un grafo conexo, no dirigido. Para éstos grafos, es posible hallar un grafo de peso mínimo que conecte todas las ciudades, conocido también como árbol generador mínimo (agm). Precisamente, ésto es lo que nos pedía el enunciado: hallar la forma de conectar todas las ciudades de forma tal que el coste de unirlas todas sea mínimo. Con nuestro planteo, logramos responder dicho problema, teniendo en cuenta, además, la complejidad pedida para resolverlo. La misma fue lograda no sólo implementando un algoritmo conocido, como es el de prim, sino que se fue cuidadoso a la hora de elegir las estructuras de datos utilizadas para mantenernos dentro de la cota estipulada. Por último, pudimos mostrar que nuestro algoritmo se comporta de manera similar para cuando la cantidad de aristas total del grafo es la misma, sin importar si éstas se consiguen agregando más vértices, o bien, agregando más aristas por vértices.

# 3. Problema 3

## 3.1. Enunciado

El problema que se nos plantea en este enunciado consiste en encontrar un camino posible para que un sapo se mueve entre piedras a diferente distancias en base a saltos, teniendo que llegar de una cierta piedra a otra, con la condicion de que cada salto tiene que estar en un rango de distancias. Más en abstracto, la idea base del problema y a partir de como modelamos nuestro escenario a partir de los datos de entrada, es encontrar el camino desde un vertice a otro de un grafo ponderado, sin necesidad de ser el camino minimo.

## 3.2. Solución

La solución propuesta, al ser un problema común de grafos, es buscar el camino de un vertice a otro, y como no es necesario que sea el camino mínimo usamos **DFS** (Busqueda en profundida), priorizando la complejidad lineal en vez del un camino eficiente y mínimo. Para esto, sabiendo que como maximo el sapo puede saltar 10 (o menos) piedras de un salto, modelamos un grafo con todos los nodos, y luego agregamos las aristas de posible salto entre cada nodo, que como maximo podrián ser de 20, un número constante. Una vez generado el grafo, se procede, mediante el algoritmo **DFS** buscar el nodo final empezando por el inicio. El algoritmo **DFS**, es un algoritmo muy conocido, con un funcionamiento ya demostrado y una complejidad proporcional a los nodos y aristas de $O(|V| + |A|)$, que dado un nodo inicial y un nodo final, busca recursivamente un camino en un grafo entre aquellos nodos, con la particuralidad de que recorre los nodos en profundidad, es decir, cuando esta recorriendo los nodos adyacentes, siempre expandirse por cada nodo que recorre, hasta que ya no queden mas nodos o haya encontrado el camino. Para nuestra solución utilizamos el algoritmo tal cual es, con la unica optimizacion inicial que comprueba si el movimiento entre vertices puede ser directo sin necesidad de seguir buscando.Entonces, volviendo al algoritmo, en cada paso recursivo, marca como visitado el nodo actual, luego comprueba si el salto puede ser directo y termina la busqueda, en caso contrario, seguirá visitando los nodos adyacentes al nodo actual que aun no fueron visitados, de un máximo de 20, haciendo recursion con **DFS** en cada uno hasta encontrar el nodo final, dando por finalizada la búsqueda.

La demostración de la correctitud del algoritmo se sustenta sobre el hecho de que el algoritmo basicamente se encarga de generar un grafo con todos los nodos y aristas de posible salto con los datos de entrada, paso necesario para poder resolver la instancia, para luego, realizar una búsqueda entre dos nodos del grafo con el algoritmo de **DFS** (Busqueda en profundida), la cual su correctitud ya esta demostrada.

Siendo G mi grafo generado a partir de los datos de entrada, el algoritmo recorrerá recursivamente la componente conexa perteneciente al nodo del cual se inicia la búsqueda, por lo tanto para que exista un camino, el nodo inicial y final deberan pertenecer a la misma componente conexa, es decir, que en caso de ser un grafo conexo (que ninguna piedra esté fuera del rango de distancias de posible salto), siempre habrá una solución para todo par de nodos. El caso base será cuando el salto entre piedras pueda ser directo, caso que siempre tendrá lugar cuando los dos nodos pertenezcan a la misma componente conexa, pero en caso de que no pertenezcan, en el paso recursivo, recorrerá en profundidad todos los nodos adyacentes de cada nodo del grafo de la componente conexa del nodo inicial llegando al resultado de que no existe un camino entre el par de nodos.

## 3.3. Pseudocódigo

```
RESOLVER (in nodos,in x,in y,in p,in q)  ⟶  pila : Pila
1  Si el salto puede ser directo                                   ▷ O(1)
2      return pila = (x,y);
3  Si no
4      grafo = Genero el grafo                                     ▷ O(n)
5      pila = Busco con DFS(grafo, inicio = x, fin = y)            ▷ O(n)
6  Devuelvo pila
```

---

```
DFS (in grafo,in inicio,in fin)  ⟶  pila : Pila
 1  Si el salto puede ser directo                                  ▷ O(1)
 2      return pila = (x,y);
 3  Si no
 4      Marco visitado al vértice inicio                           ▷ O(1)
 5      Para cada nodo adyacente del vértice inicio                ▷ O(1)
 6          Si no fue visitado                                     ▷ O(1)
 7              Si es el vértice fin que estoy buscando            ▷ O(1)
 8                  Agrego este vértice a la pila                  ▷ O(1)
 9                  Devuelvo la pila
10              Si no
11                  Lo marco como visitado
12                  pila = Busco con DFS a partir de este nodo     ▷ O(n)
13                  Agrego este vértice a la pila
14                  Devuelvo la pila
15      Devuelvo pila vacia
```

---

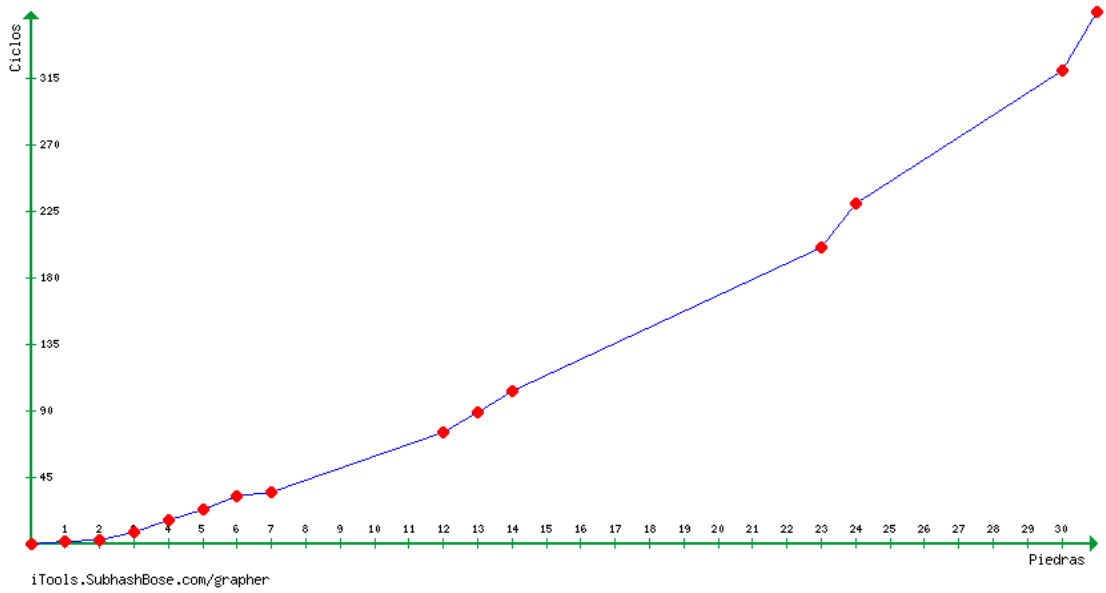
## 3.4. Analisís de complejidad

El algoritmo consta de 2 etapas:

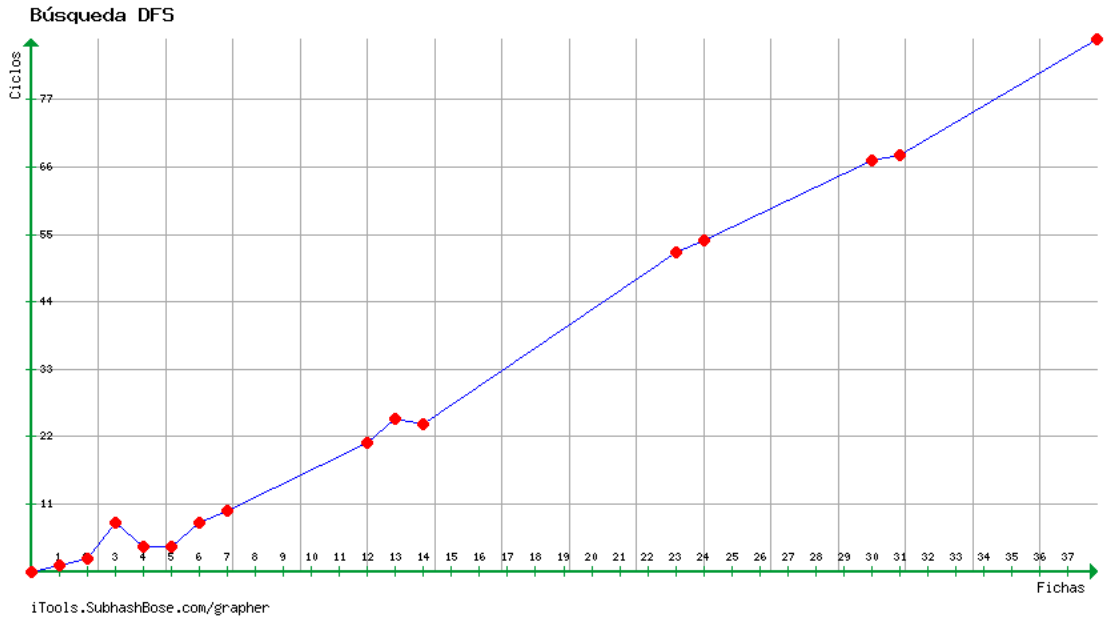
Primero se genera el grafo, agregando todas las piedras como nodos, y uniendo cada nodo con aristas solamente si el salto es posible. Al haber n nodos, y por cada nodo un maximo de 20 aristas (ya que el salto maximo puede ser de 10 piedras en cualquier *dirección*), la complejidad final y sacando las constantes quedaría en O(n)
Segundo, se debe buscar el camino, mediante el algoritmo de DFS, partiend desde la piedra de inicio hasta la final, sin repetir los nodos ya visitados, con lo cual, al ser la complejidad del algoritmo: O($|V| + |E|$), en nuestro caso pasaria a ser O(n + 10n) = O(n), debido a que las aristas de cada nodo estan acotadas por el rango de distancia de saltos que puede realizar, que no pueden ser mayor a 10
Quedando como complejidad final O(n), tal como fue pedida.
*Observación:* La cota mínima de $\Omega$ (1) se da en caso de que el salto pueda ser directo.

## 3.5. Tests y Gráficos

En el siguiente gráfico, con respecto a la cantidad de piedras, se puede observar mediante la cantidad de ciclos totales que realizó el algoritmo para encontrar una solución, que la complejidad temporal es lineal a la cantidad de piedras:




Además, enfocandonos en el algoritmo de búsqueda DFS que realizamos luego de generar nuestro grafo, se puede observar como también la complejidad es lineal en relación a la cantidad de piedras:




Con lo cual, mediante un cantidad limitada de tests, se puede observar que el algoritmo propuesto para resolver el problema se comporta de la manera correcta y con la complejidad pedida.

## 3.6. Conclusiones

Como conclusión al problema dado en relación con la complejidad pedida, se pudo modelar un grafo a partir del simulacro de un sapo con el objetivo de llegar de desde una piedra de partida a otra de llegada, saltando entre distintas piedras a diferentes distancias con saltos limitados entre rangos; de tal forma que pueda encontrar un camino posible entre nodos, priorizando la performance temporal del algoritmo en vez de una solución más eficiente. En caso contrario, si se quiere buscar un camino mínimo, la complejidad aumentaría considerablemente. A pesar de ello, pudimos demostrar que además de que el algoritmo cumple lo pedido, para diferentes instancias con rangos, piedras y caminos variados, el algoritmo se suele comportar de la misma manera con complejidad lineal a la cantidad de piedras que tenga la instancia y esto es principalmente debido al algoritmo de búsqueda en profundidad usado que como máximo visitará todos los n nodos del grafo.