



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

9 de mayo de 2014

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Chapresto, Matías	201/12	matiaschapresto@gmail.com
Dato, Nicolás	676/12	nico_dato@hotmail.com
Fattori, Ezequiel	280/11	ezequieltori@hotmail.com
Vileriño, Silvio	106/12	svilerino@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		



Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (+54 +11) 4576-3300

<http://www.exactas.uba.ar>

Índice

1. Ejercicio 1	3
1.1. Descripción del problema	3
1.2. Descripción del algoritmo	3
1.2.1. Etapa 1: Calculo de la funcion S	4
1.2.2. Etapa 2: Calculo de la funcion P	4
1.2.3. Etapa 3: Simulacion del juego	5
1.3. Correctitud del algoritmo	5
1.4. Complejidad teorica esperada	6
1.5. Verificación de correctitud	6
1.6. Análisis de performance	6
1.6.1. Consideraciones de compilacion en el analisis de performance	6
1.6.2. Experimentos	6
1.6.3. Experimentos adicionales	8
2. Ejercicio 2	10
2.1. Descripción del problema	10
2.1.1. Ejemplo	10
2.2. Ideas para la resolución	10
2.2.1. Algoritmo	11
2.3. Justificación del procedimiento	13
2.3.1. Demostración de Kruskal sobre Prim	14
2.3.2. Demostración de que aplicar Kruskal nos genera la solución	15
2.4. Cota de complejidad	15
2.5. Casos de prueba y resultado del programa	16
2.5.1. Caso aleatorio	16
2.5.2. Mejor y peor caso	18
2.6. Mediciones de performance	20
3. Ejercicio 3	23
3.1. Descripción del problema	23
3.2. Ideas para la resolución	23
3.3. Estructuras de datos	25
3.4. Algoritmos y Complejidad	26
3.4.1. Generación del Grafo	26
3.4.2. Cálculo de alcanzables	27
3.4.3. Búsqueda del camino mínimo	28
3.4.4. Construcción del camino	29
3.4.5. Complejidad Total	30
3.5. Mediciones de performance	30
4. Apéndice: Generación de casos de prueba	37
5. Apéndice: Código fuente relevante	38
5.1. Ejercicio 1	38
5.2. Ejercicio 2	40
5.3. Ejercicio 3	44
6. Apéndice: Entregable e instrucciones de compilacion y testing	49
6.1. Estructura de directorios	49
6.2. Compilacion y ejecucion	49
6.3. Generacion de tests aleatorios y toma de tiempos	49

Resumen

En este informe se explicaran los problemas planteados en el enunciado, nuestra solucion propuesta, junto a su modelo asociado, y cualquier proposicion que hayamos asumido para tomar decisiones. Luego de plantear un modelo que resuelva la solucion- en el primer ejercicio se utiliza programacion dinamica, en el segundo algoritmos para calculo de AGM, y en el tercero una variacion de algoritmos de camino minimo-, se justificara porque nuestro modelo resuelve el problema mediante un algoritmo tambien propuesto por nosotros sobre ese modelo. Mas tarde, se demostrara que el algoritmo es correcto, y que su cota de complejidad teorica es adecuada dados los requerimientos. Ademas del analisis teorico, se realizan varias pruebas empiricas, tanto de correctitud como de estimacion de complejidad con mediciones finas tomadas en base a tests aleatorios generados con distribucion uniforme y casos borde como ser mejor o peor caso para cada algoritmo segun la entrada. Finalmente, se adjunta el codigo relevante de cada ejercicio, y una seccion explicando como correr los programas, generar tests y/o tomar tiempos.

1. Ejercicio 1

1.1. Descripción del problema

El problema consiste en modelar un juego de robanúmeros en el cual ambos jugadores juegan empleando la estrategia óptima. El juego consiste en una serie de cartas con valores enteros que los jugadores deben ir robando por turnos. Solo se puede robar una serie de cartas que sean contiguas y que comiencen por alguno de los extremos. El jugador que obtenga una suma mayor gana. El algoritmo debe calcular, a partir de una tanda de cartas, como van a ser las jugadas de ambos jugadores en cada turno, suponiendo que ambos jugadores juegan de "forma óptima", es decir, de manera de maximizar la diferencia final a favor suponiendo que el contrincante hará lo mismo cuando sea su turno. En la figura 1 se muestran ejemplos del problema junto con parte de su solución. Los turnos del primer jugador están en rojo y los del segundo jugador en azul.

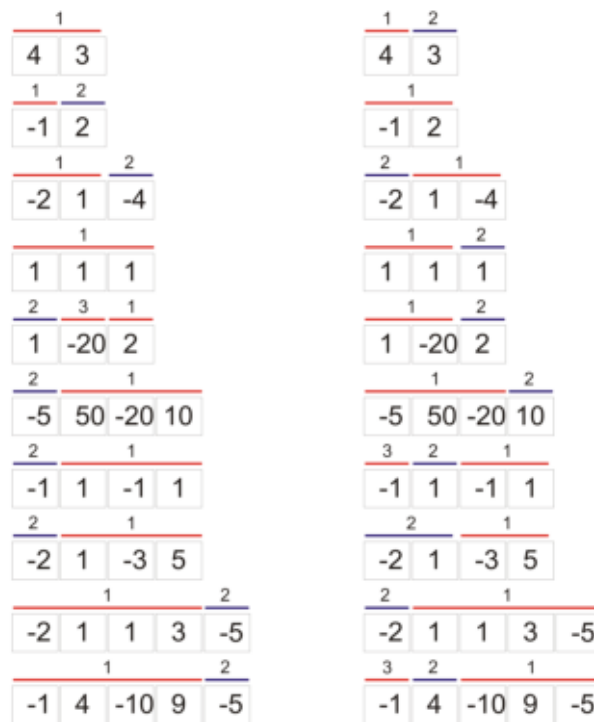


Figura 1: Soluciones óptimas del juego (izquierda) y no óptimas(derecha).

1.2. Descripción del algoritmo

Este problema presenta una estructura recursiva. Cada vez que un jugador hace su jugada, el problema queda reducido a una instancia mas pequeña del mismo, en la cual el jugador que comienza es el otro. La estrategia para resolverlo es programación dinámica.

Se llamará A_i a la tira de números inicial, con la que el juego comienza. También se define, coloquialmente, el término "juego óptimo" para referirse al desarrollo de un juego en el cual cada jugador juega según la estrategia dada por el enunciado. Esta estrategia es: jugar en cada jugada de manera de maximizar la diferencia de puntos a favor, suponiendo que el otro jugador en cada jugada hará lo mismo. El "juego óptimo" queda entonces definido recursivamente a partir del caso base: un juego que comienza con una única ficha, que solo puede jugarse de una manera (el primer jugador agarrando esa ficha).

Se definen las siguientes funciones:

$$S(i, j) = \sum_{i \leq k \leq j} A_k$$

$$P(i, j, h) = p1 - p2$$

Donde $p1$ y $p2$ son los puntajes sumados por los jugadores 1 y 2 respectivamente, luego de jugar óptimamente la subinstancia dada por la sub tira $A_k; i \leq k \leq j$, comenzando por el jugador h (jugador1 = 1, jugador2 = -1).

1.2.1. Etapa 1: Calculo de la funcion S

Se calcula la función S . Se hace por programación dinámica guardando en una tabla los valores $S(i, j)$ y usando la fórmula de recursión

$$S(i, j) = S(i, j - 1) + A_j.$$

(con $S(i, i - 1) = 0$).

```

for i=1 to n do
  for j=i to n do
     $S(i, j) = S(i, j - 1) + A_j$ 
  end for
end for

```

1.2.2. Etapa 2: Calculo de la funcion P

Se calcula la función P , también por programación dinámica, usando la S ya calculada. Se guarda en una tabla $P(i, j, 1)$, pudiendo calcularse $P(i, j, -1) = -P(i, j, 1)$ a partir de ésta. La regla de recursión es:

$$P(i, j, 1) = \max\{ \max\{S(i, k) + P(k + 1, j, -1) \mid i \leq k \leq j\}, \max\{S(k, j) + P(i, k - 1, -1) \mid i \leq k \leq j\} \}$$

con $P(i, i - 1, -1) = 0$.

Ya que la recursión requiere para el cálculo de $P(i, j)$ el cálculo de $P(i, j')$; $j' \leq j$ y de $P(i', j)$; $i' \geq i$ la tabla se calcula entonces de menor a mayor en las columnas, y por cada columna de mayor a menor en las filas.

La tabla contiene en el lugar (i, j) la tupla $(res(i, j), P(i, j))$ donde $res(i, j)$ son los índices que limitan la sub tira que queda por jugar despues de jugar la sub tira (i, j) . Si ya no se juega mas, $res(i, j) = (0, 0)$.

```

1: for j=1 to n do
2:   for i=j to 1 do
3:      $MAX = -\infty$ 
4:     for k =i-1 to j do
5:       if  $S(i, k) + P(k + 1, j) > MAX$  then
6:          $MAX \leftarrow S(i, k) + P(k + 1, j)$ 
7:         if  $k \neq j$  then
8:            $res(i, j) \leftarrow (k + 1, j)$ 
9:         else
10:           $res(i, j) \leftarrow (0, 0)$ 
11:        end if
12:      end if
13:    end for
14:    for k = j+1 to i do
15:      if  $S(k, j) + P(i, k - 1) > MAX$  then
16:         $MAX \leftarrow S(k, j) + P(i, k - 1)$ 

```

```

17:         if  $k \neq i$  then
18:              $res(i, j) \leftarrow (i, k - 1)$ 
19:         else
20:              $res(i, j) \leftarrow (0, 0)$ 
21:         end if
22:     end if
23: end for
24: end for
25: end for

```

1.2.3. Etapa 3: Simulacion del juego

Se calcula, a partir de las tablas ya calculadas, el desarrollo del juego a partir de su instancia inicial. Se guarda la cantidad de turnos jugados y las fichas robadas por izquierda o derecha en cada turno.

```

var  $turno \leftarrow 1$ 
var  $puntj1 \leftarrow 0$ 
var  $puntj2 \leftarrow 0$ 
var  $i \leftarrow 1$ 
var  $j \leftarrow n$ 
var  $jugadas(lado, cant)$ 
loop
    if  $turnos = impar$  then
         $puntj1 \leftarrow puntj1 + puntoslevantados$ 
    else
         $puntj2 \leftarrow puntj2 + puntoslevantados$ 
    end if
    if  $levantoporzq$  then
         $jugadas.lado[turno - 1] \leftarrow izq$ 
         $jugadas.cant[turno - 1] \leftarrow cantcartaslevantadas$ 
    else
         $jugadas.lado[turno - 1] \leftarrow der$ 
         $jugadas.cant[turno - 1] \leftarrow cantcartaslevantadas$ 
    end if
    if  $res(i, j) = (0, 0)$  then
        break
    end if
     $(i, j) \leftarrow res(i, j)$ 
     $turno \leftarrow turnos + 1$ 
end loop

```

Donde $puntoslevantados$, $cantcartaslevantadas$ y $levantoporzq$ se obtienen como función $O(1)$ de $res(i, j)$, y no se explicitan para simplificar la exposición. Las variables $puntj1$, $puntj2$, $turno$ y $jugadas$ se usan luego para construir la salida.

1.3. Correctitud del algoritmo

A continuación se da una demostración de por qué la estrategia elegida cumple el requerimiento del enunciado. Este requerimiento es: "maximizar la diferencia de puntos a favor al final del partido, asumiendo que el otro jugador tiene la misma estrategia".

La estrategia elegida es: para cada posible tira que se puede extraer, calcular la suma de los números de la tira y sumarle la diferencia que se consigue a favor al jugar de forma óptima durante el resto del juego (que ya está definida por ser el resto del juego una instancia más pequeña). Extraer la tira que maximice este número.

Sea entonces A_k una tira inicial de cartas. Si $A_k = (c1)$ hay una única manera de jugar, y por lo tanto la estrategia cumple el requerimiento. Supóngase que la estrategia cumple el requerimiento si A_k es

de longitud $\leq m$. Para A_k de longitud $m + 1$, una forma genérica de jugar el juego es una elección de cartas a robar inicialmente, de puntuación total S , seguida de un cierto desarrollo en el cual la diferencia conseguida a favor es P . Llámese P' a la diferencia a favor conseguida como resultado de jugar óptimamente el juego que comienza a partir de la primer elección de cartas (comenzado por el jugador2). Por hipótesis inductiva se cumple que $P \leq P'$ luego la diferencia a favor obtenida en este juego es $S + P \leq S + P'$. Además, por definición, $S + P' \leq M$ donde M es la diferencia a favor que se consigue haciendo la primer elección según la estrategia, y jugando óptimo durante el resto del juego.

Por último resta ver que esta estrategia se traduce formalmente en la recursión dada para $P(i, j, h)$ en la descripción del algoritmo, pero esto es muy fácil de ver por inducción y no se escribe.

1.4. Complejidad teorica esperada

La parte 1 del algoritmo es $O(n^2)$, ya que son 2 bucles for anidados con interior $O(1)$ con a lo sumo n iteraciones cada uno. La parte 2 del algoritmo es $O(n^3)$ ya que son 2 bucles for anidados con a lo sumo n iteraciones cada uno, y dentro de ellos hay dos bucles for sucesivos, cada uno de ellos $O(n)$. La parte 3 del algoritmo es $O(n)$ ya que el loop se ejecuta a lo sumo tantas veces como la cantidad de turnos, que es a lo sumo n , y el interior del loop es $O(1)$.

Por lo tanto la complejidad temporal total del algoritmo es $O(n^3)$

1.5. Verificación de correctitud

Para la verificación se emplearon las mismas instancias que fueron dadas como ejemplo del problema y su resolución. Éstas fueron calculadas a mano, y luego se verificó que efectivamente el algoritmo daba el mismo resultado (la misma diferencia de puntos a favor) en cada caso.

1.6. Análisis de performance

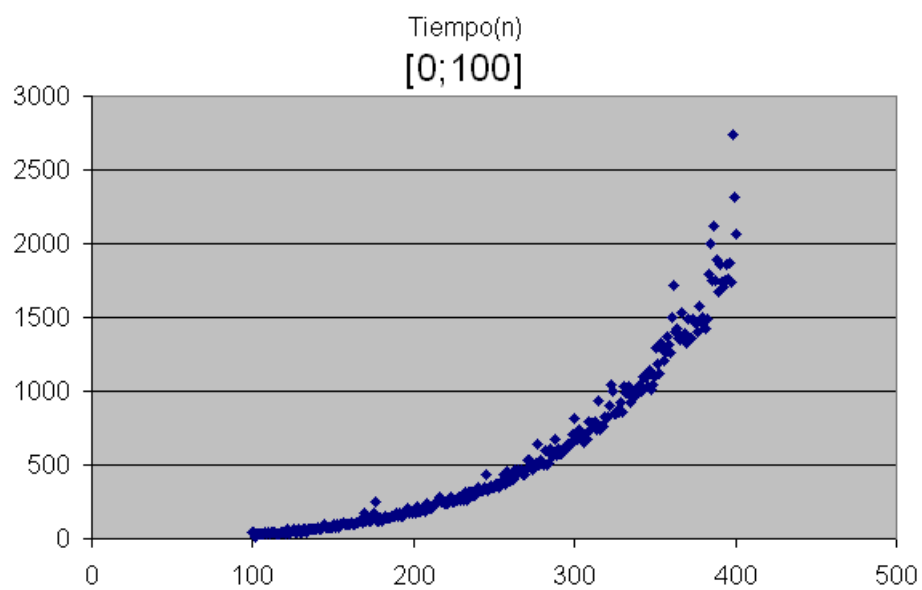
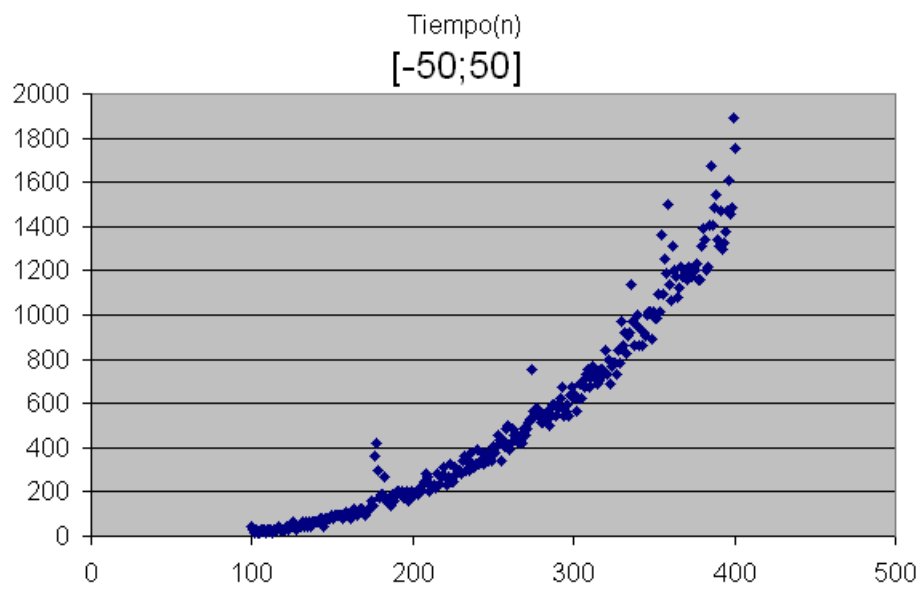
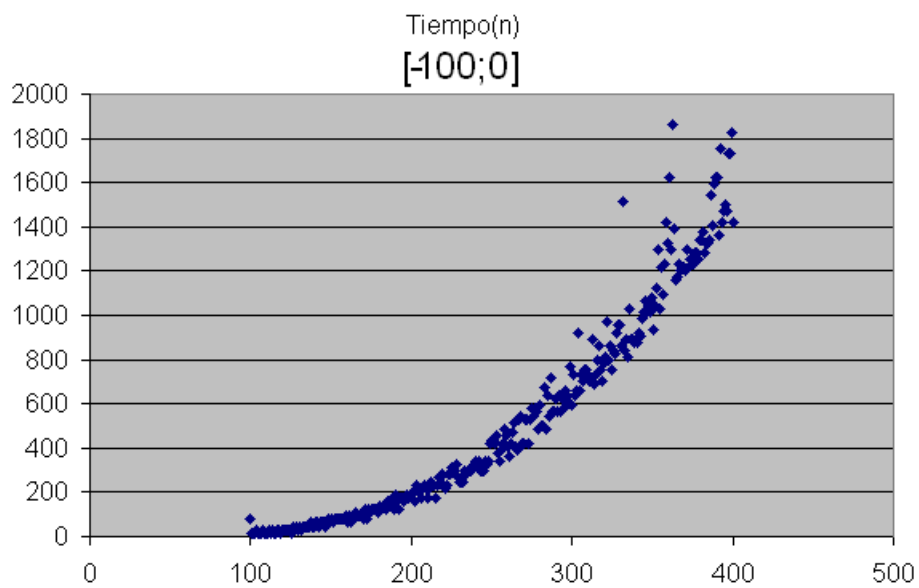
1.6.1. Consideraciones de compilacion en el analisis de performance

Compilador GJC: Java realiza un profiling sobre el programa a compilar y toma estadísticas para realizar diferentes procesos de compilacion/interpretacion sobre el codigo, esto se ve reflejado en el tiempo de ejecucion del programa compilado con el jdk, para forzar la compilacion de java a codigo objeto, utilizamos el compilador GCJ de GNU, que es uno de los compiladores denominados Ahead-Of-Time. Todos los experimentos fueron ejecutados sobre el programa compilado a codigo nativo. La cantidad de iteraciones por cada entrada es 100 para licuar los posibles outliers.

1.6.2. Experimentos

La segunda parte del algoritmo, el cálculo de la tabla P , es de complejidad $O(n^3)$ y se lleva la mayor parte del tiempo de ejecución para n grande. El cálculo de esta tabla, que tiene $n^2/2$ posiciones, involucra para cada posición el cálculo del máximo de una cantidad de elementos que depende únicamente de la posición que se calcula, no del contenido de las cartas. Como el tiempo que demora el algoritmo está dominado por esta parte, es de esperar que el contenido de las cartas no cambie apreciablemente los tiempos de ejecución. Ésto se puede constatar por medio de la experimentación.

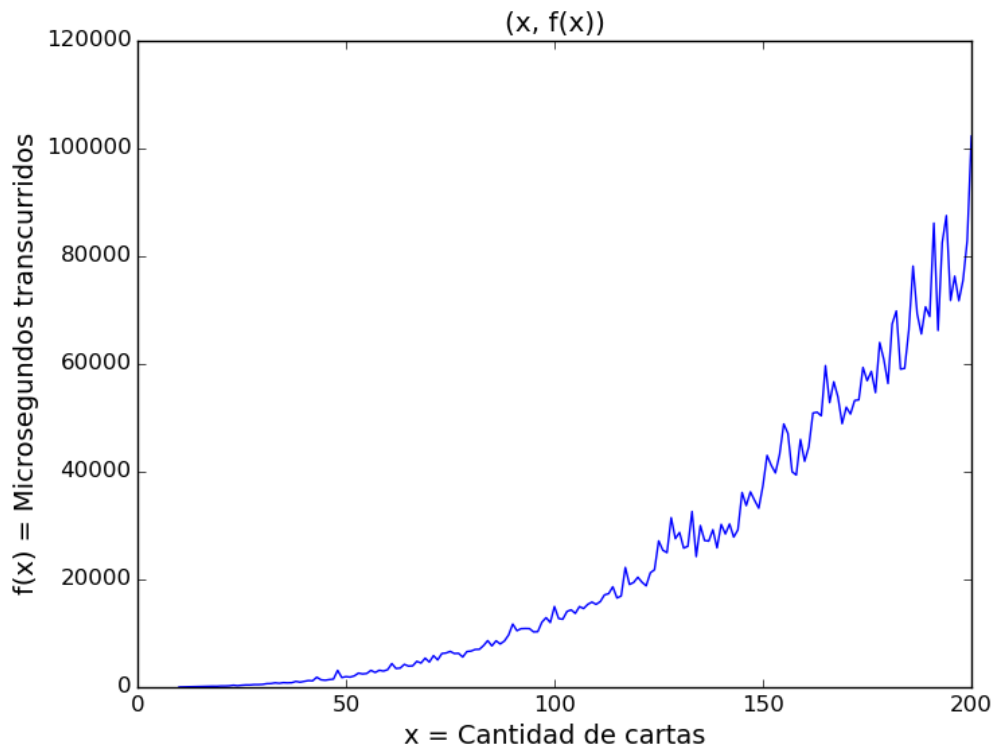
Se hizo una experimentación con tamaños n de entrada entre 100 y 400. Se distinguió la distribución de los valores de las cartas en tres casos : uniforme en $[-50; 50]$, uniforme en $[-100; 0]$ y uniforme en $[0; 100]$. Todos los casos dieron tiempos similares. En la gráfica puede observarse una curva similar a las de la forma ax^m . Se puede comprobar que al variar de $n = 200$ a $n' = 300$, $n' = (3/2)n$, el tiempo pasa de un promedio de 200 a un promedio de $650 \sim 200 \cdot (3/2)^3$. Ésto comprueba que la curva es cúbica. A continuación se adjuntan las gráficas experimentales de $tiempo(n)$.

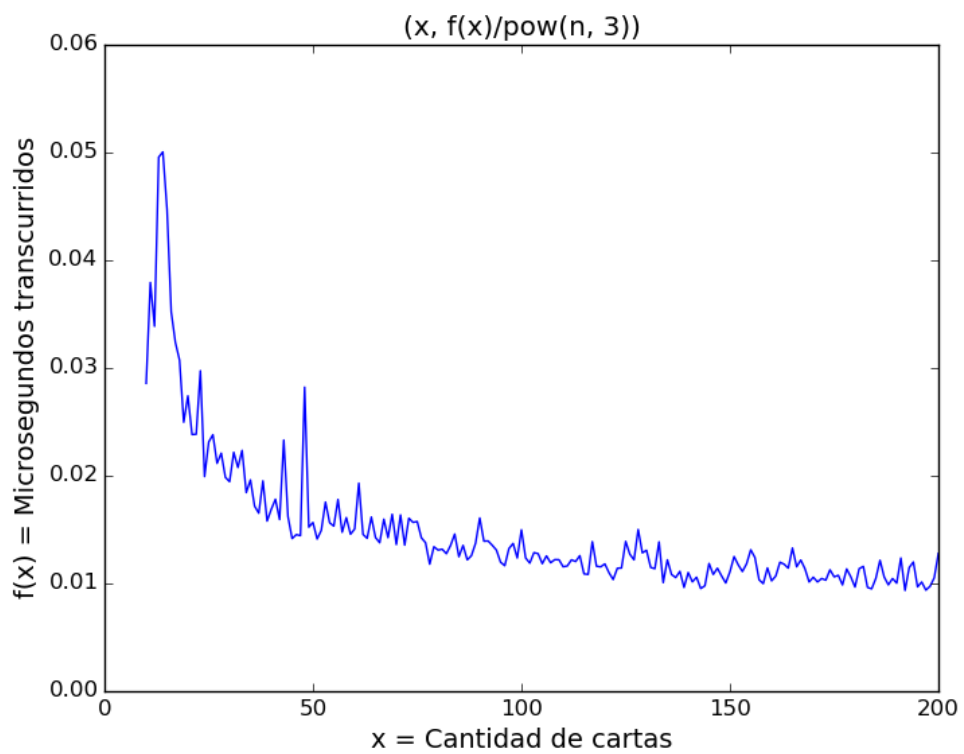
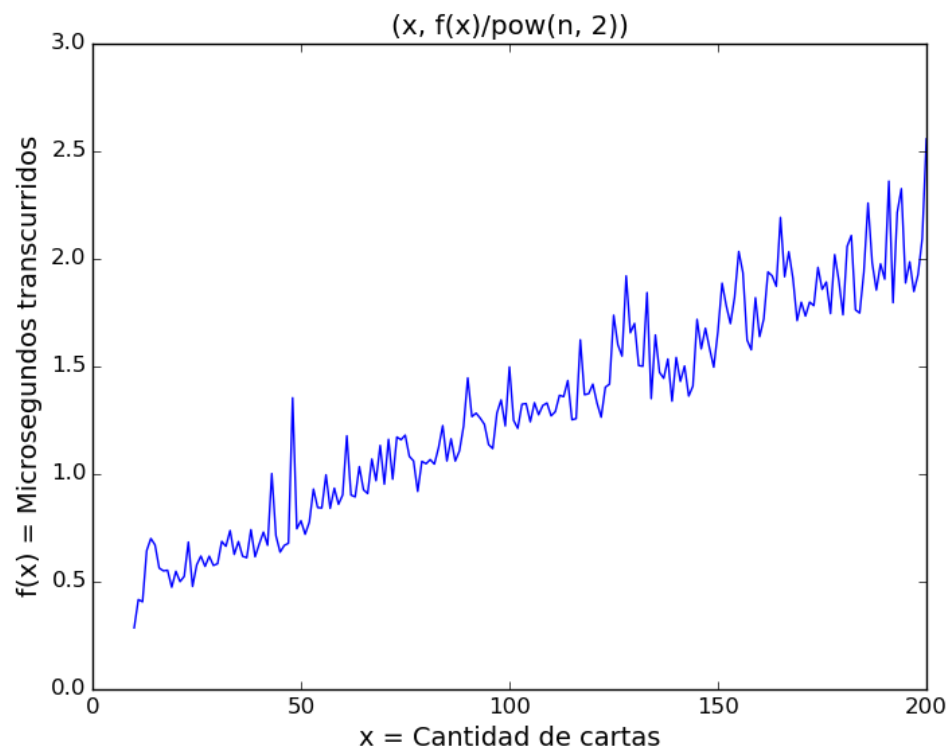


1.6.3. Experimentos adicionales

Se realizaron ademàs experimentos adicionales con el objetivo de verificar de forma practica que la complejidad graficada sera $O(n^3)$.

Interpretacion de los graficos: Como se puede ver en los graficos, la complejidad empirica, coincide con la complejidad teorica esperada $O(n^3)$, llegamos a esta conclusion graficando los casos: $n = [1 \dots 200]$ para cada caso, se realizaron 3 graficos, $(X, F(X))$, $(X, F(X)/x^2)$ y $(X, F(X)/x^3)$. En estos graficos se puede observar que al dividir por x^2 se obtiene una curva con crecimiento lineal aproximado, y al dividir por x^3 , se obtiene una constante aproximada, es decir, una curva acotada entre una diferencia pequena en el eje X, concluyendo que la curva inicial corresponde a una funcion $f(n) = n^3$, concluyendo, $f(n) \in O(k * n^3)$.





2. Ejercicio 2

2.1. Descripción del problema

El problema se trata de un conjunto de ciudades ubicadas a cierta distancia entre ellas, las cuales todas deben de ser provistas de gas. Para ésto, tenemos una cantidad k de centrales distribuidoras de gas, y tuberías para conectar ciudades. Una ciudad tiene gas si hay un camino de tuberías que llegue hasta una central distribuidora, es decir, si una ciudad está conectada a otra ciudad por una tubería y a su vez ésta está conectada a otra ciudad por medio de una tubería la cual tiene una central distribuidora, entonces las 3 ciudades tienen gas. Se pide lograr que todas las ciudades tengan gas, pero que la longitud de la tubería más larga de la solución debe ser la más corta posible. La longitud de una tubería es igual a la distancia entre las 2 ciudades.

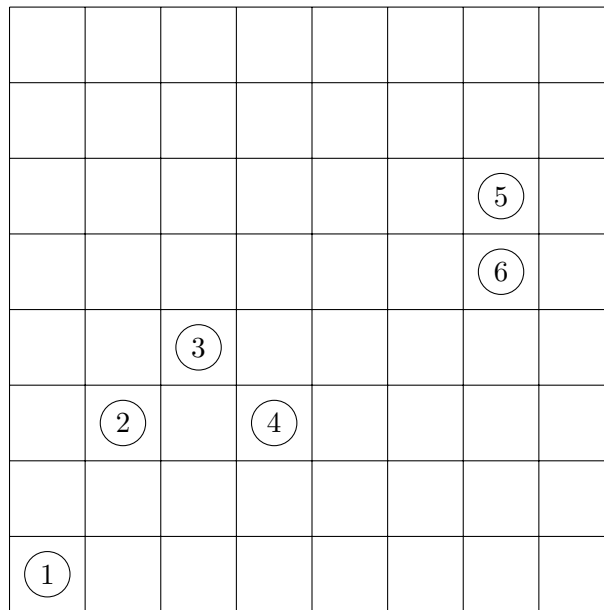
El algoritmo tiene que tener una complejidad de $O(n^2)$, con n la cantidad de ciudades.

De a partir de ahora, k va a ser siempre la cantidad de centrales distribuidoras disponibles, y n la cantidad de ciudades.

En todos los gráficos se va a colocar una grilla donde cada casillero de la grilla representa una unidad de distancia.

2.1.1. Ejemplo

Como entrada podemos tener 6 ciudades y 2 centrales distribuidoras, como en la Figura 2, y la solución que se busca es como indica la Figura 3.



Se tienen 2 centrales disponibles

Figura 2: La entrada del problema, con las 5 ciudades y la cantidad de centrales

2.2. Ideas para la resolución

Para la resolución del problema, se puede pensar en un *grafo*, donde las ciudades son nodos y las tuberías las aristas, cada arista va a tener una distancia asociada que es la distancia entre los nodos (ciudades) que conecta.

Como cada ciudad tiene gas si hay un camino hasta una central distribuidora, entonces todas las ciudades que se conectan a una misma central pertenecen a una misma componente conexas. Como tenemos un máximo de k centrales, la solución tiene que tener un máximo de k componentes conexas y las centrales se colocarán cada una en una componente conexas y en cualquier ciudad dentro de la componente conexas, ya que la distancia de las aristas dentro de una componente conexas no se verá modificada si se cambia de nodo la central dentro de la misma componente conexas.

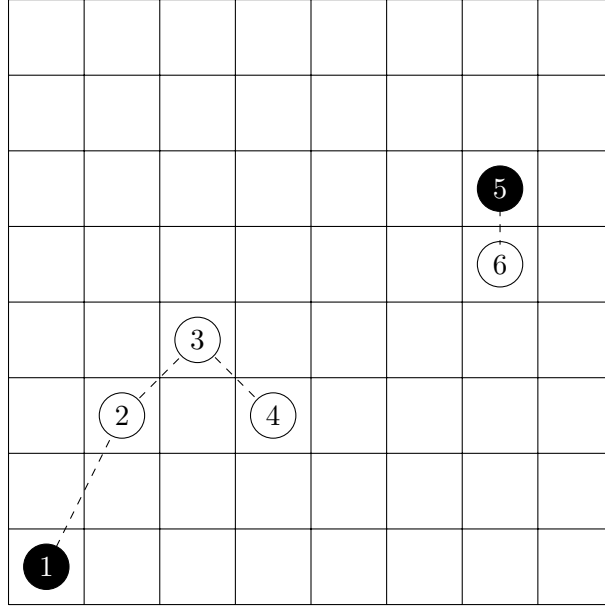


Figura 3: La solución al problema de la Figura 2, los nodos negros son los que contienen las centrales de distribución

Entonces, la solución que se pide es que el grafo tenga a lo sumo k componentes conexas, y que la distancia de la arista mas larga, sea la más corta posible.

Una idea que se propone es comenzar con todos los nodos sueltos, sin ninguna arista conectada, y calcular todas las distancias entre todos los nodos, es decir, calcular todas las tuberías posibles con sus respectivas distancias. Luego se ordenarán las aristas (tuberías) de menor a mayor de acuerdo a sus longitudes. Se calcula si la cantidad de componentes conexas es menor o igual a k , si es así, entonces el algoritmo termina, sino coloca la tubería más corta que no genere un ciclo y continúa preguntando sobre la cantidad de componentes conexas e iterando sobre las aristas siguiendo el orden de menor a mayor. Básicamente es aplicar un algoritmo parecido a *Kruskal* sobre un grafo completo.

Como para aplicar la idea anterior se requiere tener todas las aristas ordenadas por peso, en un grafo completo tenemos $n(n - 1)/2$ aristas, dandonos una complejidad de al menos $O(n^2 \log n^2)$ para ordenarlas, y no cumple con el requerimiento. Para mejorar la complejidad del algoritmo, en vez de calcular la distancia de todas las aristas e iterar sobre todas las aristas, requiriendo ordenar por peso *todas* las aristas, primero creamos un árbol generador mínimo (con las aristas más cortas posibles), con una idea como el algoritmo de *Prim* y que tenga de cota $O(n^2)$, el árbol generador mínimo generado nos queda $n - 1$ aristas, y luego, al igual que antes, se ordenan y se recorren esas $n - 1$ aristas de menor a mayor.

En resumen, la idea es partir de un grafo completo con n nodos, aplicar *Prim* y obtener el *AGM*, luego sobre ese árbol aplicar *Kruskal* pero cortando el algoritmo cuando se tienen tantas componentes conexas como k centrales de gas. La colocación de las centrales de gas se colocan luego en un nodo cualquiera de los nodos de cada componente conexa.

En la sección 2.2.1 se propone un pseudocódigo, en la sección 2.3 se justificará la correctitud, y en la sección 2.4 se hará el cálculo de la complejidad del algoritmo.

2.2.1. Algoritmo

El algoritmo propuesto lo que realiza es generar un *AGM* siguiendo al algoritmo de *Prim* (desde la Línea 5 a 30), y luego se ordenan las aristas del *AGM* para ir agregándolas siguiendo la idea de *Kruskal* y detenerse cuando se llega a k componentes conexas (desde la Línea 31 a 34).

Algorithm 1 minimizarTuberias

Require: *centrales*: cantidad de centrales disponibles, mayor que 0

Require: *ciudades*: las ciudades con sus posiciones

Require: *n*: cantidad de ciudades en el parámetro *ciudades*

Ensure: Retorna el grafo tal que hay tanas componentes conexas como *centrales*, y que la arista más larga es la más corta posible

```
1: procedure MINIMIZARTUBERIAS(Entero: centrales, Array: ciudades, Entero: n)→ Grafo
2:   Grafo g ← NUEVOGRAFO(n)                                ▷ Creo un grafo con n nodos, sin aristas
3:   <bool agregado, entero distancia, entero nodo> nodos[n] ▷ La distancia es desde n (índice del
   array) hasta nodos[n].nodo
4:   <nodo1, nodo2, distancia> aristas[n - 1]

5:   nodos[0] ← <true, 0, 0>
6:   for i ← 1; i < n; i++ do
7:     nodos[i] ← <false, DISTANCIA(ciudades[i], ciudades[0]), 0>
8:   end for

9:   for agregados ← 0; agregados < n - 1; agregados++ do
10:    distancia_minima ← ∞
11:    nodo_minimo ← 0
12:    for i ← 0; i < n; i++ do                                ▷ Busco el nodo mas cercano al árbol que ya tenemos
13:      if nodos[i].agregado = false then
14:        if nodos[i].distancia < distancia_minima then
15:          distancia_minima ← nodos[i].distancia
16:          nodo_minimo ← i
17:        end if
18:      end if
19:    end for

20:    nodos[nodo_minimo].agregado ← true
21:    aristas[agregados] ← <nodo_minimo, nodo[nodo_minimo].nodo, distancia_minima>    ▷
   Agrego el nodo encontrado

22:    for i ← 0; i < n; i++ do                                ▷ Actualizo la distancia de los nodos no agregados aún
23:      if nodos[i].agregado = false then
24:        if nodos[i].distancia > DISTANCIA(ciudades[i], ciudades[nodo_minimo]) then
25:          nodo[i].distancia ← DISTANCIA(ciudades[i], ciudades[nodo_minimo])
26:          nodo[i].nodo ← nodo_minimo
27:        end if
28:      end if
29:    end for
30:  end for

31:  ORDENAR(aristas)                                            ▷ Ordeno las aristas por la distancia

32:  for componentes ← n; componentes > centrales; componentes-- do
33:    AGREGARARISTA(g, aristas[n - componentes].nodo1, aristas[n - componentes].nodo2)    ▷
   AgregarArista está especificado en el Algoritmo ??
34:  end for
35:  retornar ← g
36: end procedure
```

Una vez que tenemos el árbol, lo que tenemos que hacer es seleccionar un nodo cualquiera de cada componente conexas para colocar la central distribuidora. Para ésto al crear un nodo sin aristas, cada nodo se va a asociar a una componente conexas distinta, y cada vez que se agrega una arista, se unen las

componentes conexas de ambos nodos bajo una misma componente. Luego al tener el grafo armado, se recorre la lista de componentes conexas y se agarra un nodo para cada componente:

Algorithm 2 AgregarArista

```

1: procedure AGREGARARISTA(Grafo g, Nodo n1, Nodo n2)
2:   NUEVAARISTA(g, n1, n2)      ▷ agrega la arista entre n1 y n2, sin actualizar las componentes
   conexas
3:   if COMPONENTECONEXA(g,n1) ≠ COMPONENTECONEXA(g,n2) then
4:     nueva ← COMPONENTECONEXA(g,n1)
5:     vieja ← COMPONENTECONEXA(g,n2)
6:     for n ∈ NODOS(g) do      ▷ Le asigno a todos los nodos de la componente conexa vieja, la
   nueva componente conexa
7:       if COMPONENTECONEXA(g,n) = vieja then
8:         SETEARCOMPONENTE(g,n,nueva)
9:       end if
10:    end for
11:  end if
12: end procedure

```

Y por último para obtener un Nodo para cada componente conexa, vamos a crear un array con n elementos, cada posición i representa a la componente conexa i , y adentro puede tener un valor nulo representando que esa componente conexa no existe, o el valor de un nodo, entonces se va a retornar para cada componente conexa existente, un nodo, para que esos nodos representen las centrales:

Algorithm 3 NodosDeComponentes

```

1: procedure NODOSDECOMPONENTES(Grafo g)Nodo[CantidadNodos(g)]
2:   Nodos nodos[CantidadNodos(g)]
3:   for i = 0; i < CantidadNodos(g); i++ do ▷ Inicializo todas las componentes conexas (que son
   como máximo igual a la cantidad de nodos) asignandole ningún nodo
4:     nodos[i] = nil
5:   end for
6:   for n ∈ g do                  ▷ para cada nodo, le asigno dentro del
   array nodos tomando como índice la componente conexa el valor del nodo. Así al finalizar el ciclo
   sólo las componentes conexas existentes y con nodos tienen un valor dentro del array nodos, y va a
   tener el valor del último nodo correspondiente a esa componente conexa, ya que nodos de la misma
   componente se van pisando en el array
7:     nodos[COMPONENTECONEXA(g,n)] ← n
8:   end for
9:   return nodos
10: end procedure

```

2.3. Justificación del procedimiento

Para la justificación vamos a justificar primero que si se aplica el algoritmo de *Kruskal* sobre el *AGM* generado por el algoritmo de *Prim* dado un grafo completo, entonces llegamos a un grafo con pesos de aristas iguales que aplicar *Kruskal* sobre el grafo completo directamente, aunque puede tener aristas diferentes de igual peso. Como tienen los mismos pesos aunque las aristas sean diferentes, a efectos de minimizar el peso de la arista de mayor peso da el mismo resultado. (Sección 2.3.1).

Luego vamos a justificar que aplicando el algoritmo de *Kruskal* sobre un grafo completo pero cortando cuando tenemos K componentes conexas, nos genera la solución que buscamos (Sección 2.3.2).

Teniendo demostrado esos 2 puntos, entonces aplicar *Kruskal* a un grafo completo nos da la solución que buscamos y aplicar *Kruskal* al resultado *Prim* (que es lo que realiza el algoritmo propuesto) nos da el mismo resultado que aplicar *Kruskal* directamente, siendo la solución buscada.

2.3.1. Demostración de Kruskal sobre Prim

Demostraremos que aplicar el algoritmo de *Kruskal* sobre un grafo completo da aristas con los mismos pesos que aplicar *Kruskal* sobre el *AGM* resultante del algoritmo de *Prim*. Como el problema pone restricción sobre el peso de las aristas (de las tuberías), entonces nos fijaremos que las aristas del resultado pueden ser diferentes pero si se mira sólo los pesos de las aristas, estos son los mismos.

Demostración. Un *AGM* de un grafo tiene como suma de los pesos de sus aristas, la menor suma posible de todos los árboles de un grafo. Si se aplica *Prim* y *Kruskal* a un mismo grafo (un grafo completo por ejemplo), ambos dan un *AGM* como resultado, por ende el peso del árbol generado por *Prim* del grafo g ($P_p(g)$) es igual al peso del que genera *Kruskal* ($P_k(g)$), si no fueran iguales, el que tiene mayor peso no sería un *AGM*. Como ámbos son árboles del mismo grafo, ambos tienen $n - 1$ aristas ($n =$ cantidad de nodos del grafo).

$$\begin{aligned} P_p(g) &= \sum_{i=0}^{i < n-1} \pi(a_i) \\ &= \\ P_k(g) &= \sum_{i=0}^{i < n-1} \pi(a_i) \end{aligned}$$

Como *Prim* y *Kruskal* son 2 algoritmos diferentes, podrían generar un árbol con diferentes aristas pero igual mantienen la igualdad en la suma de los pesos y la cantidad de aristas.

Supongamos que un algoritmo pone las mismas aristas que el otro pero en vez de poner la arista a pone la arista a' con distinto peso, entonces también tiene que haber cambiado otra arista b por b' para mantener la igualdad de la suma. Supongamos que tenemos todas las aristas a_i del *AGM* y las ordenamos de menor a mayor y de la arista a_0 a a_{j-1} y de a_{j+2} a a_{n-2} las aristas son las mismas para ambos algoritmos pero a_j y a_{j+1} son diferentes, entonces:

$$\begin{aligned} P_p(g) &= \sum_{i=0}^{i < j} \pi(a_i) + \pi(a') + \pi(b') + \sum_{i=j+2}^{i < n-1} \pi(a_i) \\ &= \\ P_k(g) &= \sum_{i=0}^{i < j} \pi(a_i) + \pi(a) + \pi(b) + \sum_{i=j+2}^{i < n-1} \pi(a_i) \end{aligned}$$

Simplificamos la igualdad de las suma, nos queda:

$$\begin{aligned} \pi(a') + \pi(b') \\ &= \\ \pi(a) + \pi(b) \end{aligned}$$

Como dijimos que el peso de a' es distinto de a , podemos suponer que $\pi(a) > \pi(a')$ y por lo tanto $\pi(b) < \pi(b')$ (se puede suponer también lo contrario y sería análogo). *Prim* entonces hubiera agregado a' con menor peso que a , pero como *Kruskal* recorre las aristas de menor a mayor peso, ya tendría que haber pasado por a' , y como las aristas anteriores a a' son las mismas en ambos algoritmos, agregar a' no genera ciclos, entonces *Kruskal* hubiera agregado a a' cuando llegó el momento de agregar a , entonces $\pi(a') = \pi(a)$ porque $a' = a$. Aún quedaría $\pi(b') \neq \pi(b)$, pero como todas las demás aristas tienen el mismo peso, entonces para mantener la igualdad de la suma nos queda que $\pi(b') = \pi(b)$, contradiciendo la suposición que un algoritmo cambiaría una arista por otra con peso distinto.

Si cambia una arista por otra de igual peso, no modifica la solución porque el peso de la arista de mayor peso seguiría siendo el mismo.

Como aplicar *Kruskal* a un árbol da el mismo árbol (porque es el único subgrafo que es un árbol), aplicar *Kruskal* al resultado de *Prim* da el mismo resultado que aplicar directamente *Prim*, y ya demostramos que aplicar *Prim* y *Kruskal* dan árboles con mismos pesos de aristas, por lo que aplicar *Kruskal* directamente o aplicar *Kruskal* a la salida de *Prim*, da árboles con los mismos pesos de aristas. \square

2.3.2. Demostración de que aplicar Kruskal nos genera la solución

Demostraremos que teniendo un grafo completo, si aplicamos *Kruskal* y dejamos de agregar aristas cuando llegamos a k componentes conexas, tenemos la solución que buscamos:

Demostración. El algoritmo de *Kruskal* ordena las aristas de menor a mayor según su peso, y agrega 1 a 1 las aristas según dicho orden tal que no genere un ciclo. Al agregar una arista se disminuye en 1 la cantidad de componentes conexas. Nosotros detendremos el algoritmo cuando se lleve a k componentes conexas. Supongamos que tenemos m aristas ordenadas de menor a mayor por su peso, $(a_0, a_1 \dots a_{m-1})$, que se fueron agregando (aunque algunas generaban ciclos y no se agregaron al *AGM*), y que detuvimos el algoritmo de *Kruskal* al agregar la arista a_j porque se llegaron a k componentes conexas. Entonces, como las aristas estaban ordenadas:

$$\begin{aligned} (\forall 0 \leq i < j) a_i &\leq a_j \\ &\wedge \\ (\forall j < i < m) a_i &\geq a_j \end{aligned}$$

El a_j es la arista de mayor peso (la tubería más larga), y es la que queremos minimizar.

Si existiese una mejor forma de armar las componentes conexas, entonces existe un $a_i, a_i < a_j \implies i < j$, tal que se puede reemplazar a_i por a_j y así tener la arista de mayor peso que es de menor peso que la que obtuvimos. Como $i < j$, *Kruskal* ya la analizó a_i antes de pasar por a_j .

Si a_i no generaba ciclos, entonces esa arista pertenece al resultado del algoritmo de *Kruskal*, y como no se detuvo en a_i significa que aún agregando esa arista (y todas las anteriores) no se llegaban a las k componentes conexas que se buscan, entonces no se puede sacar a_j porque no se llegaría a la cantidad de componentes conexas necesarias, contradiciendo que se puede quitar a_j para mejorar la solución.

Si a_i generaba un ciclo, entonces esa arista no pertenece al resultado del algoritmo. Notar que como a_i se analiza antes que a_j por tener menor peso, entonces a_i generaba un ciclo cuando a_j aún no estaba agregada. Si se quita a_j y se agrega a_i , al quitar a_j se aumenta en 1 la cantidad de componentes conexas (ahora tenemos $k + 1$) y no es solución porque no hay tantas centrales para poner en cada componente, y al agregar a_i se está generando un ciclo, por lo que no disminuye la cantidad de componentes conexas, manteniéndose en $k + 1$ y no es solución tampoco, por lo que contradice que se puede quitar a_j para agregar a_i y tener una mejor solución.

Si se detuviera el algoritmo al obtener $k - 1$ componentes conexas, ésta tendría 1 arista más que si se tienen k componentes conexas, y como se agregan en orden entonces esa arista nueva es mayor o igual que teniendo k componentes, por lo que haría de mayor peso la arista más larga, no siendo óptima la solución.

Entonces, agregar aristas en orden según su peso si no genera ciclos y parar cuando se alcanzan las k componentes conexas, nos da como peso de la arista más larga, el peso mas chico posible, siendo la solución que se busca del problema. \square

2.4. Cota de complejidad

Se analiza la complejidad de algoritmo 1, para ésto se supone que el cálculo de la distancia entre ciudades (el peso de las aristas/tuberías) se realiza en $O(1)$

La cantidad de nodos se representará como n , y la cantidad de centrales como k .

- La inicialización de unas variables se realiza de la línea 5 a 8, y se realizan n iteraciones calculando la distancia y guardando en un vector, quedando $O(n)$

- Para la creación del AGM entre las líneas 9 a 30 realiza $n - 1$ iteraciones
 - Líneas 12 a 19, busca el nodo más cercano, realizando n iteraciones de comparaciones y asignaciones que son $O(1)$, quedandonos $O(n)$
 - Las líneas 20 y 21 son $O(1)$
 - La actualización de las distancias, entre las líneas 22 y 29 realiza n iteraciones de comparaciones y asignaciones que son $O(1)$, quedandonos $O(n)$
- Ordenar las aristas en la línea 31, y se realizan sobre las aristas agregadas en la iteración anterior, la cual agrega $n - 1$ aristas, entonces se ordenan $n - 1$ elementos. Al n no estar acotado, se puede realizar con algoritmos como *MergeSort* o *HeapSort* en $O(n \log n)$
- Entre las líneas 32 y 34 se realizan $n - k$ iteraciones, implementando el grafo g en una *matriz de adyacencia*, *AgregarArista* se realiza en $O(n)$ ya que agrega la relación en la matriz de adyacencia en $O(1)$ y luego tiene que actualizar las componentes conexas en $O(n)$ quedandonos una complejidad de $O(n * (n - k))$. Notar que si $k > n$, no se realiza ninguna iteración.
- Luego de tener el grafo solución, tenemos que obtener 1 nodo de cada componente, eso se realiza con la función *NodosDeComponentes* que inicializa un array de n elementos y luego recorre todos los nodos (n nodos) una vez, quedando $O(n + n) = O(n)$

Bajo éste análisis, la complejidad nos queda (1)

$$\begin{aligned}
 &O(n) + (n - 1) * (O(n) + O(1) + O(n)) + O(n \log n) + O(n * (n - k)) + O(n) \\
 &= \\
 &O(n) + O(n^2) + O(n) + O(n^2) + O(n \log n) + O(n * (n - k)) + O(n) \\
 &= \\
 &O(3n) + O(2n^2) + O(n \log n) + O(n * (n - k)) \\
 &= \\
 &O(n^2) + O(n * (n - k))
 \end{aligned} \tag{1}$$

Como el $O(n * (n - k))$ está acotado por $O(n^2)$ (cuando $k = 0$), entonces nos queda que la complejidad del algoritmo es:

$$O(n^2)$$

Cumpliendo así la complejidad requerida.

2.5. Casos de prueba y resultado del programa

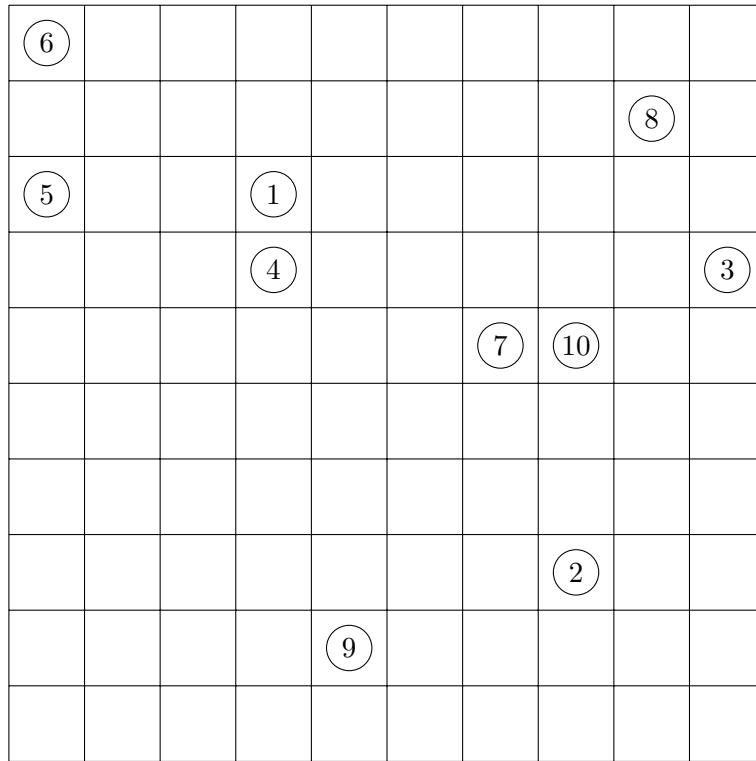
Se generaron casos aleatorios, casos de peor y mejor entrada teniendo 10 ciudades y se corrió el algoritmo sobre dichas entradas obteniendo su solución.

2.5.1. Caso aleatorio

Para el caso aleatorio, se utiliza el programa descrito en la Sección 4. Se obtuvo una entrada con 10 ciudades dispuesta aleatoriamente en un espacio de 10 x 10, y con k centrales, k un valor aleatorio entre 1 y 10

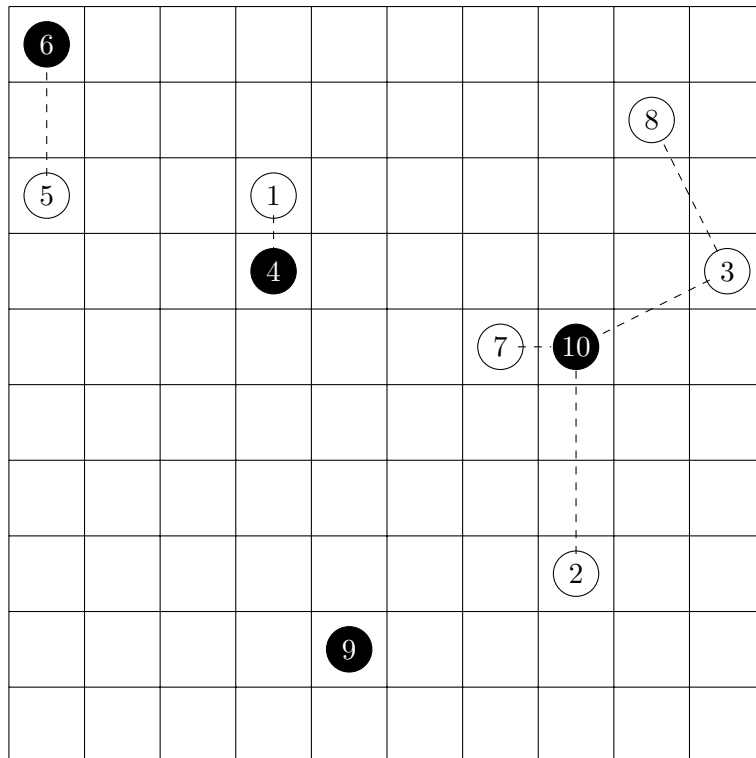
Entrada: Figura 4

Solución: Figura 5



Se tienen 4 centrales disponibles

Figura 4: La entrada del problema, con las 10 ciudades y la cantidad de centrales



Se tienen 4 centrales disponibles

Figura 5: Solución del problema de la Figura 4 dado por el algoritmo propuesto, con las 10 ciudades y las 4 centrales hubicadas en los nodos oscuros

2.5.2. Mejor y peor caso

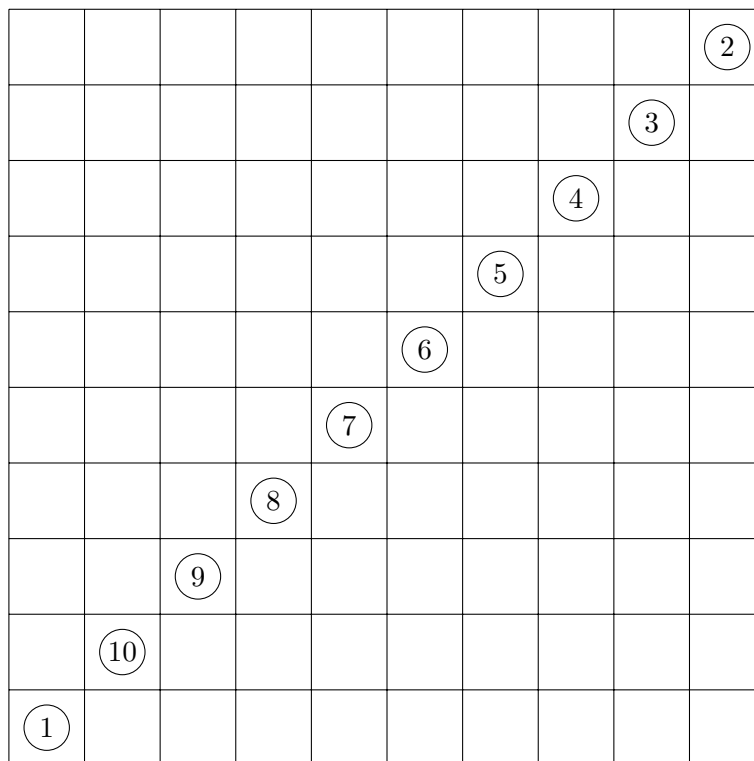
Dados 2 casos con las mismas ciudades, el algoritmo realizará más operaciones en la que tenga menos centrales disponibles porque tiene que agregar más tuberías, ésto se ve en la complejidad calculada en la Sección 2.4, donde nos quedaba $O(n^2) + O(n * (n - k))$, si $k = n$, entonces queda $O(n^2)$, y si $k = 1$, queda $O(n^2) + O(n * (n - 1))$.

A su vez, cuando se elige el nodo más cercano para agregar al *AGM* de *Prim* (Línea 12 del Algoritmo 1), si los nodos tienen la distancia de mayor a menor, entonces siempre entra a la condición de actualizar la mínima distancia. Y cuando se agrega un nuevo nodo y se actualiza la nueva distancia de los nodos restantes (Línea 22 del Algoritmo 1) si siempre el nodo agregado está más cerca que los nodos que ya estaban agregados, entonces siempre se actualiza.

Entonces, el peor caso se puede ver cuando $k = 1$ y si los nodos están alineados y comenzando de la ciudad 1, el resto de las ciudades están ordenadas de mayor a menor, así el algoritmo comienza agregando el nodo 1 y luego recorre el resto de los nodos para buscar el más cercano comenzando por el más lejano y siempre que agregue un nuevo nodo (el más cercano al 1) va a tener que actualizar la distancia de todos los nodos.

El mejor caso se da cuando $k = n$ y los nodos están alineados, todos a la misma distancia de los nodos del que tienen al lado, y los números de las ciudades están ordenadas pero con la ciudad 1 (por la que se comienza) está en el medio. Así como están ordenados y a la misma distancia, siempre agarra el más cercano en el primer nodo que analiza, y por como agrega ese nodo (giamos a la derecha del 1), los que están a la izquierda no tienen que actualizar la distancia (y como el 1 está en el medio, esos son la mitad).

Entrada de peor caso: Figura 6



Se tiene 1 central disponible

Figura 6: Caso de peor entrada con las 10 ciudades y 1 central

Solución de peor caso: Figura 7

Entrada de mejor caso: Figura 8

Solución de mejor caso: Figura 9

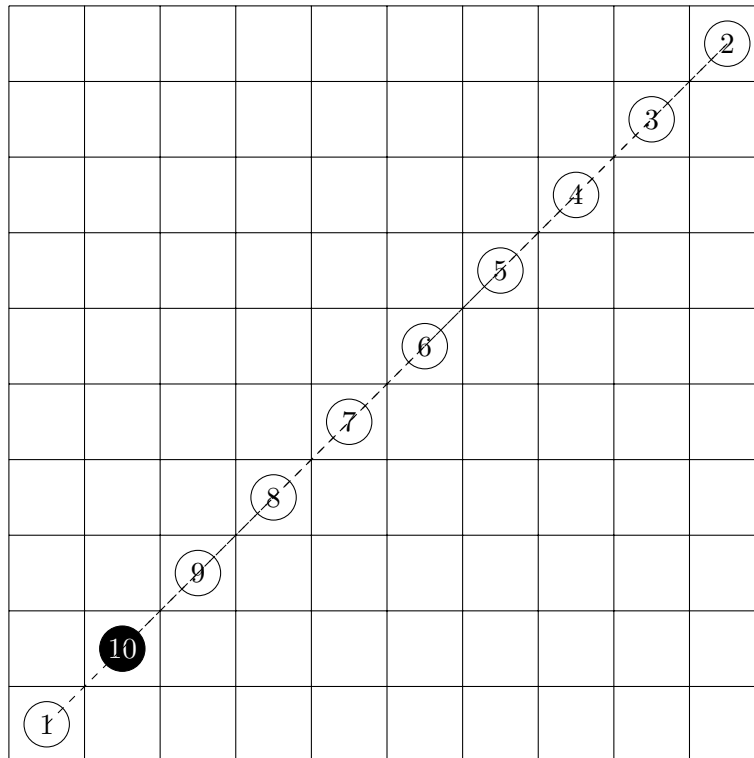
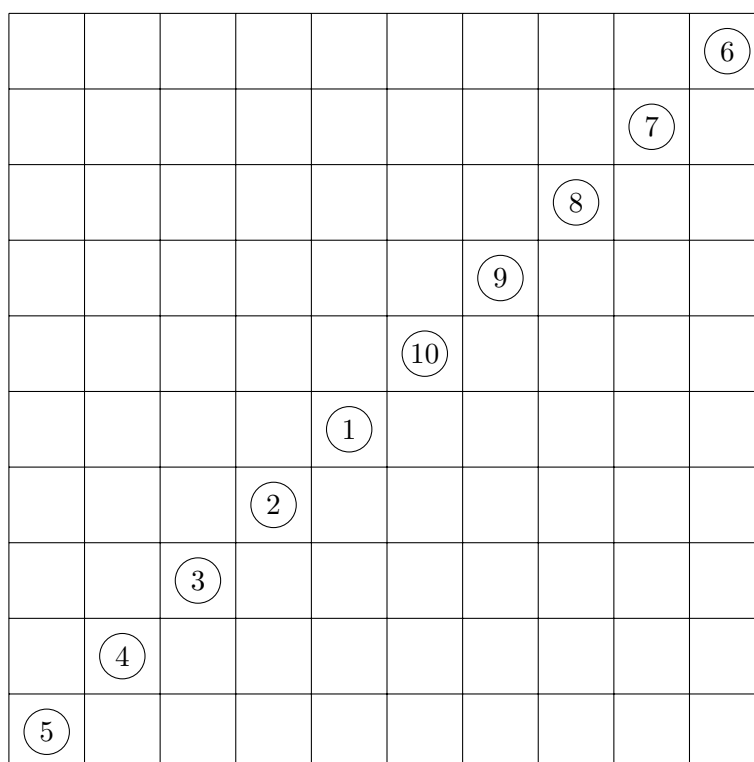


Figura 7: Solución del caso de peor entrada de la Figura 6



Se tienen 10 centrales disponibles

Figura 8: Caso de mejor entrada con las 10 ciudades y 10 centrales

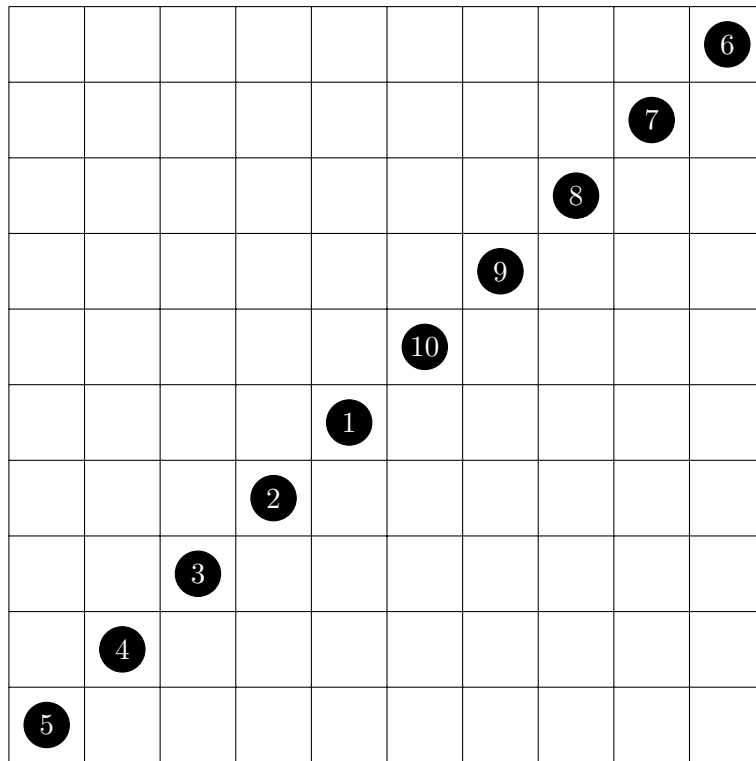
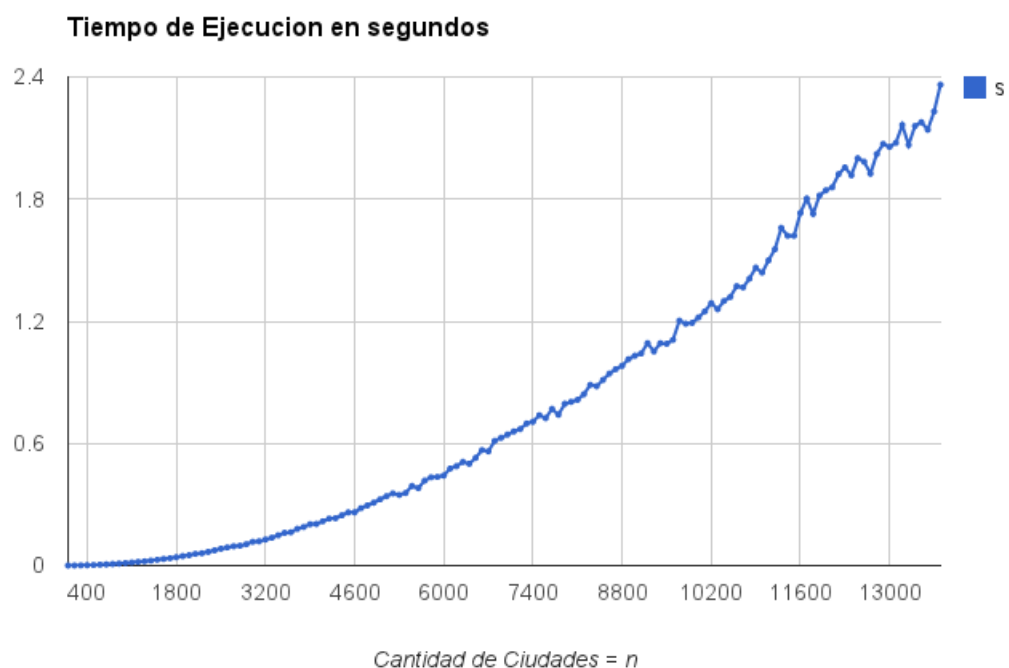


Figura 9: Solución del caso de mejor entrada de la Figura 8

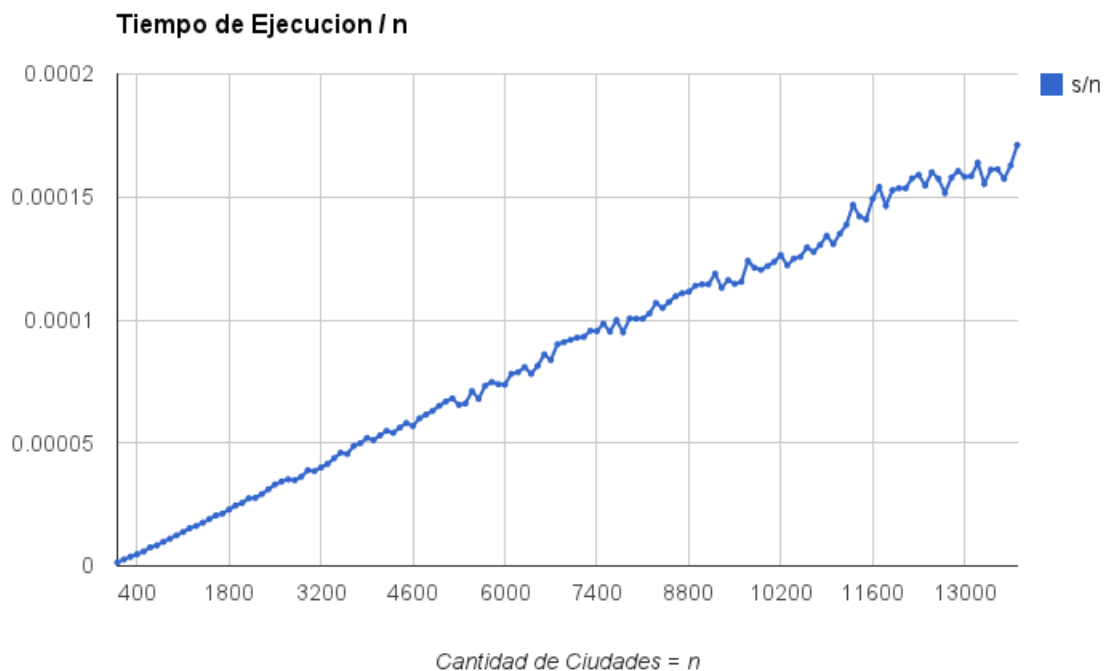
2.6. Mediciones de performance

Para medir la performance del algoritmo, se crearon casos de prueba como se especifica en el Apéndice de casos de prueba (Sección 4) y se comparó contra la complejidad $O(n^2)$ calculada en la Sección 2.4

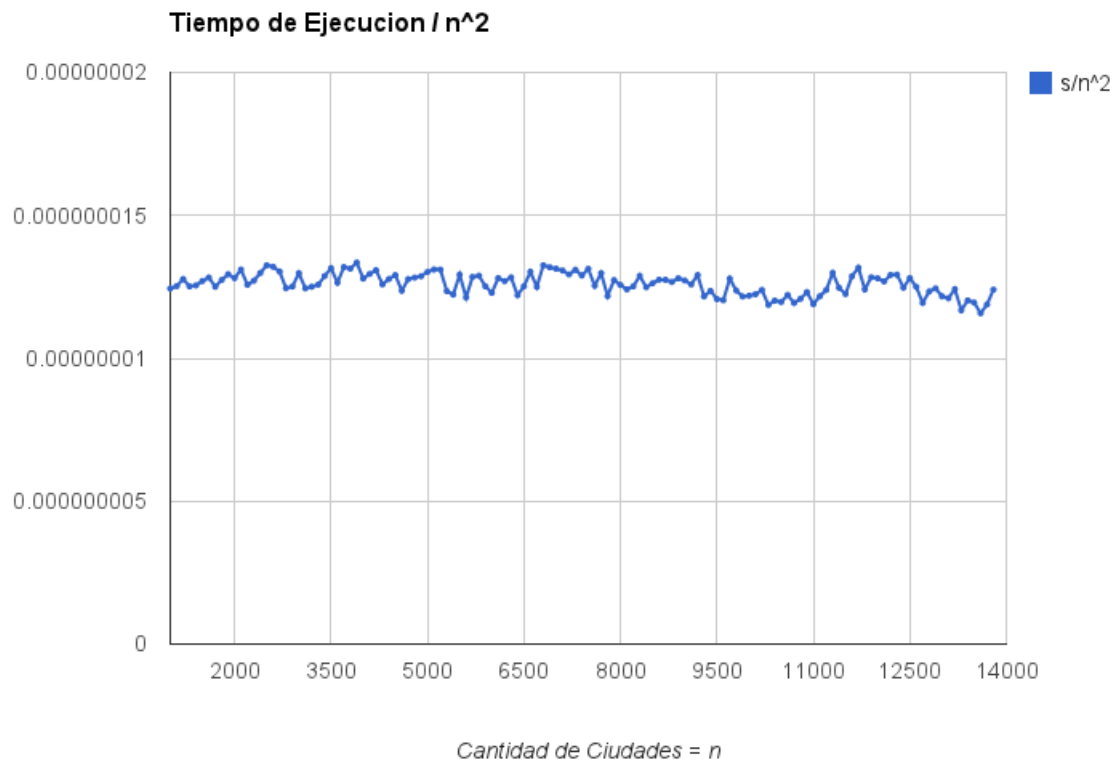
Se probó la implementación con n desde 100 a 13800, avanzando de a 100, y se graficó el promedio del tiempo en segundos que tardó el algoritmo repitiendo la prueba 10 veces.



Luego se graficó el resultado anterior diviendo el tiempo por n , quedando lo que parece una recta



Y por último se graficó el resultado anterior diviendo el tiempo por n^2 , mostrando una recta que se aproxima a una constante acotada por arriba y abajo. Se quitó del gráfico las mediciones para valores chicos de n por dar valores demasiado chicos y variables.



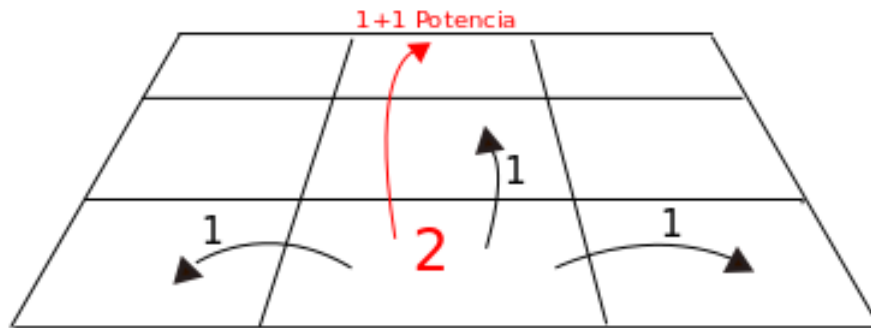
No se pudieron hacer pruebas para n mucho mas grandes por el tiempo que se tardaba en sacar las mediciones y porque el consumo de memoria crecía en el orden de n^2 , con $n = 20000$ se requerían al menos $(20000b)^2 * 4b = 16000000000b = 1600000kb = 1600mb$ de memoria (la multiplicación por 4 es por el tamaño de un entero) y al llegar a esos valores, la computadora que se usaba para las mediciones comenzaba a ponerse muy lenta porque el sistema mientras tanto movía memoria en RAM al disco rígido.

3. Ejercicio 3

3.1. Descripción del problema

El problema se trata de un juego que forma parte de un reality show. Este juego tiene un campo de juego cuadrado dividido en celdas, de tamaño $n \times n$. Cada una de estas celdas posee un resorte con el que se puede saltar hacia otras celdas. Estos resortes varían en sus potencias, siendo la potencia la cantidad de celdas que pueden propulsar al jugador. Es posible apuntar los resortes hacia adelante, atrás, izquierda y derecha. Adicionalmente cada jugador posee unidades extra de potencia, que le permiten potenciar sus saltos. Éstas son limitadas, y el jugador las podrá usar a elección en los saltos que le parezcan convenientes.

Un ejemplo de salto sería la siguiente situación: vemos que la celda tiene una potencia de 1, por lo que el jugador podría saltar una celda izquierda, hacia adelante o hacia la derecha. Y en caso de que quisiera usar una potencia, podría saltar dos celdas hacia adelante, pero su reserva de potencias se reduciría en uno.

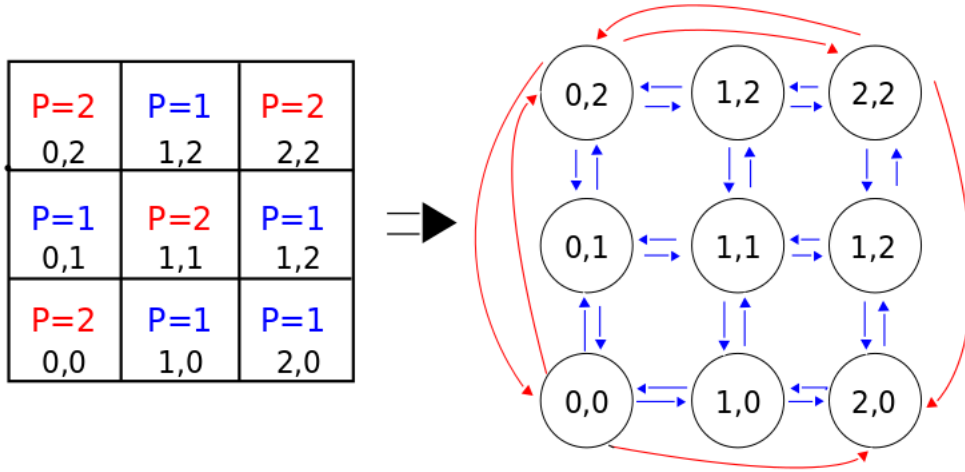


El objetivo del juego es llegar de una celda origen a otra destino, realizando la menor cantidad de saltos posibles. Por esto, se pide el diseño e implementación de un algoritmo que calcule y de como resultado uno de los caminos que lleguen de origen a destino cuya cantidad de saltos sea mínima.

El algoritmo tiene que tener una complejidad de $O(n^3 * k)$, con n la cantidad de filas y columnas del campo de juego y k la cantidad de potencias otorgadas al inicio del juego al jugador.

3.2. Ideas para la resolución

Decidimos encarar la resolución como un problema de grafos. El campo de juego es representado como un grafo con un nodo por cada celda, y las aristas son orientadas, de peso constante y representan todos los posibles saltos entre celda y celda.



Por cada uno de los valores $0..k$ de potencia tenemos un campo de juego de los anteriormente mencionados. Los vamos a llamar "*niveles*", y van a ser subgrafos interconectados para formar un grafo general que va a resolver el problema. Entre estos niveles va a haber aristas conectando los nodos de mayor nivel con los de menor nivel, representando el hecho de "*gastar*" potencias, es decir, si en un turno estoy en el nivel k y utilizo dos potencias, en el siguiente turno voy a estar posicionado en un nodo del nivel $k - 2$.

De esta forma, estar en el nodo (i, j) del nivel k representa estar posicionado en la celda de juego (i, j) y disponer de k potencias para utilizar. Podemos "*saltar entre celdas*", moviéndonos de un nodo a otro usando las aristas este nivel. En caso de querer usar las potencias, debemos movernos por aristas que conectan los nodos de nivel k con los niveles inferiores. Por ejemplo, si estamos en el nodo (i, j, k) , de potencia 2, y queremos saltar 3 nodos hacia la derecha, debemos tomar la arista orientada que nos lleva al nodo $(i + 3, j, k - 1)$ (en caso de disponer de una potencia). De esto deducimos que las aristas entre niveles sólo salen de nodos de nivel superior hacia nodos de nivel inferior. Formalicemos esto:

Para cada nivel, sea:

$$G_L = (V_L, E_L) / V_L = V_1, \dots, V_n = \{\text{casillas } 1..n \text{ con } L \text{ potencias disponibles}\}$$

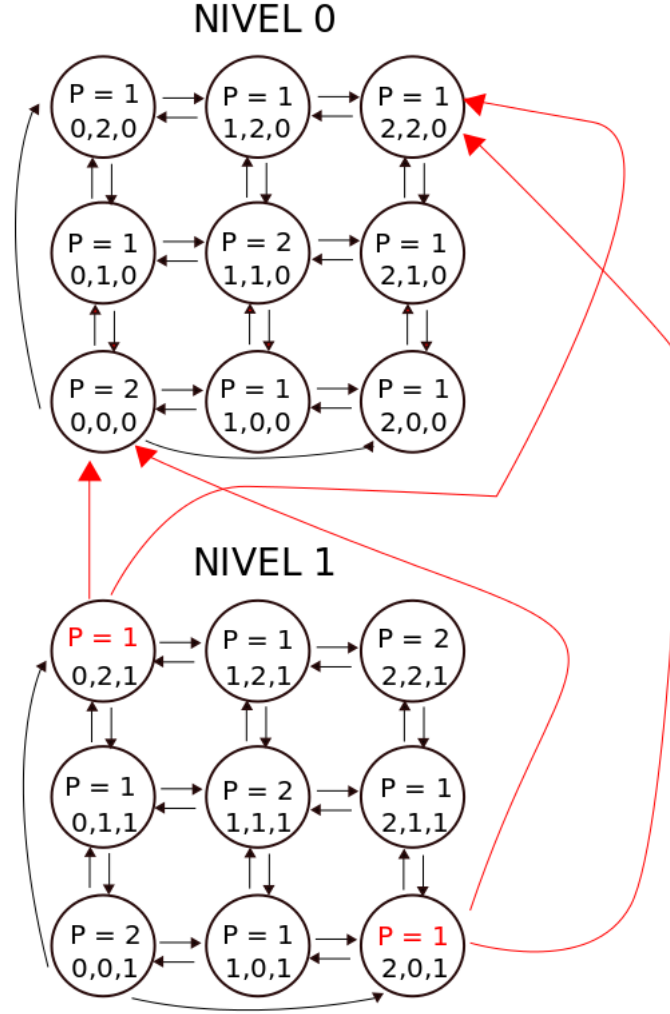
Por lo tanto, el grafo G es:

$$G = (W, A) / W = \cup_{i=1}^k V_i(G_i) \wedge A = (\cup_{L=1}^k E_L(G_L)) \cup T$$

Donde T son las aristas entre niveles, descriptas a continuación:

Se tienen aristas dirigidas en cada nivel tal que si $v, u \in V_t$, $(u, v) \in E_t$ sii v es alcanzable desde u sin usar potencias. Dados niveles $G_m, G_j / m < j$ y dos vértices $v \in V_m(G_m) \wedge w \in V_j(G_j)$ existe una arista dirigida "*entre*" niveles, que pertenezca a T , sii w es alcanzable desde v usando estrictamente $(j - m)$ unidades de potencia, es decir, que la distancia en casillas (verticales u horizontales) de v a w es igual a $(j - m) + \text{pot}(v)$.

En el siguiente diagrama mostramos las aristas entre niveles que deberían tomarse en caso de usar una potencia en las celdas $(0, 2)$ y $(2, 0)$



Notemos que al avanzar hacia niveles más bajos no se puede ir hacia arriba (lo cual tiene sentido pues el nivel actual indica la cantidad de potencias disponibles para usar, y este valor nunca puede aumentar)

Definimos n_0 como el nodo origen $/n_0 \in V_k$ (primer nivel, más alto), dado por las coordenadas cartesianas de la celda de comienzo de la entrada. Si la celda destino en el juego es (i, j) definimos el conjunto de nodos destino como los nodos (i, j, k) , con k entre 0 y la cantidad de potencias máxima, de forma que al llegar al nodo destino (i, j, l) , l es la cantidad de potencias sin usar.

Las aristas son de peso constante en su totalidad (en particular, 1). Además, en cada arista se guarda un valor indicando la cantidad de niveles atravesadas en este paso, lo cual es equivalente a la cantidad de potencias utilizadas en el salto. No confundir este valor con el costo o peso de la arista. (Dibujo zarpado)

De esta forma, encontrar el camino que realice la menor cantidad de saltos se reduce a buscar el camino más corto en el grafo de n_0 a algún nodo destino, guardando en cada caso el valor de cada arista (potencia usada) y la posición (i, j) del nodo sin importar el nivel actual. Un recorrido de anchura BFS del grafo nos brinda el camino más corto, comenzando la búsqueda desde n_0 .

3.3. Estructuras de datos

A cada una de las celdas, las representamos con la clase *nodo*, que posee tres atributos: x , y y $level$, (el numero de fila y columna de la celda que representa este nodo, y el nivel en el que se encuentra el nodo).

Representamos el grafo en memoria con una modificación del método de listas de adyacencia. La estructura es un diccionario (java *HashMap*) llamado *graph*, cuyas claves son los nodos y sus definiciones son el tipo *NodoMetadata*.

La clase *NodoMetadata* brinda la información necesaria de cada nodo para poder armar el grafo, ya que contiene el atributo *alcanzables*, que es su lista de adyacencia. Además de esto, posee tres atributos más: *nodoValue*, *fueVisitado* y *predecesor*, el primero representando la potencia del resorte de la celda, y los otros dos son de utilidad al bfs, para "marcar" un nodo, e indicar su antecesor en el recorrido de éste.

Finalmente tenemos una clase *Juego* que unifica todo, cuyos atributos son un nodo inicial, un nodo destino, y el diccionario *graph*.

3.4. Algoritmos y Complejidad

El algoritmo consta de varias etapas principales:

1. Generación del grafo a partir de los datos de entrada.
2. Búsqueda del camino mínimo(en cantidad de pasos) con un BFS.
3. Construcción del camino a través de los predecesores del destino indicados por BFS y cálculo de potencia utilizado en cada salto.

3.4.1. Generación del Grafo

A continuación mostramos un pseudocódigo de la generación del juego. Durante las demostraciones, se usará la variable *k* para referirse a la cantidad de potencias iniciales al iniciar el juego, y la variable *n* como la cantidad de filas del juego.

```

1: procedure JUEGO(int[][]valores, int potInicial, int filInicial, int colInicial, int filDestino, int colDestino)
2:   Juego.nodoInicial  $\leftarrow$  new Nodo(filInicial, colInicial, potInicial)  $\triangleright O(1)$ 
3:   Juego.nodoDestino  $\leftarrow$  new Nodo(filDestino, colDestino, 0)  $\triangleright O(1)$ 
4:   Juego.graph  $\leftarrow$  new HashMap < Nodo, NodoMetadata > ()  $\triangleright O(1)$ 
5:   int dimension  $\leftarrow$  longitud(valores)  $\triangleright O(1)$ 
6:   for int level  $\leftarrow$  potInicial; level  $\geq$  0; level -- do  $\triangleright O(potInicial)$ 
7:     for int i  $\leftarrow$  0; i < dimension; i ++ do  $\triangleright O(dimension)$ 
8:       for int j  $\leftarrow$  0; j < dimension; j ++ do  $\triangleright O(dimension)$ 
9:         Nodo nodo  $\leftarrow$  new Nodo(i, j, level)  $\triangleright O(1)$ 
10:        int nodoValue  $\leftarrow$  valores[i][j]  $\triangleright O(1)$ 
11:        if nodoValue > dimension then
12:          nodoValue  $\leftarrow$  dimension
13:        end if
14:        boolean fueVisitado  $\leftarrow$  false  $\triangleright O(1)$ 
15:        List < Nodo > alcanzables  $\leftarrow$  calcularAlcanzables(nodo, nodoValue, dimension)
 $\triangleright O(dimension)$ 
16:        NodoMetadata metaData  $\leftarrow$  new NodoMetadata(nodoValue, fueVisitado, alcanzables, null)
 $\triangleright O(1)$ 
17:        graph.put(nodo, metaData)  $\triangleright O(1)$ 
18:      end for
19:    end for
20:  end for
21: end procedure

```

Donde *valores* representa la matriz de celdas con sus respectivas potencias de resorte, *potInicial* representa *k*, la cantidad de potencias que se le brinda a un jugador al iniciar el juego, y *filInicial*,

colInicial, *filDestino*, *colDestino* las coordenadas respectivas de la celda de inicio y la celda destino. Cabe aclarar, en la línea 11, en caso de que en la entrada, una celda posea un valor de resorte mayor a la dimensión del campo de juego, fijamos el valor del nodo como *dimension*, ya que no tiene sentido poseer un resorte que permita saltar fuera de los límites de la matriz, es decir, el jugador sólo va a poder usar uso de la potencia del resorte como máximo hasta el valor *dimension*. Esto nos permite ajustar el cálculo de complejidad de la función siguiente.

Complejidad:

Los 3 ciclos for anidados indican una complejidad de $O(k * n^2)$. Dentro de estos ciclos se realizan operaciones, todas de tiempo constante $O(1)$ excepto la llamada a la función *alcanzables*, que toma $O(n)$, y será justificado a continuación, dándonos un total de $O(k * n^3)$ en la construcción del grafo para nuestro modelo.

3.4.2. Cálculo de alcanzables

Función auxiliar que para un nodo, calcula todos los otros nodos a los que podría "saltar", usando y no usando potencias. La salida del algoritmo es del tipo *Lista < Nodo >*.

```

1: procedure CALCULARALCANZABLES(Nodo nodo, int valorNodo, int dimension)
2:   int nivelActual = nodo.getLevel()                                ▷  $O(1)$ 
3:   int actualX = nodo.getX()                                        ▷  $O(1)$ 
4:   int actualY = nodo.getY()                                        ▷  $O(1)$ 
5:   int resorte = valorNodo                                          ▷  $O(1)$ 
6:   Lista < Nodo > alcanzables = new Lista < Nodo >                  ▷  $O(1)$ 
7:   for int j ← -resorte; j ≤ resorte; j ++ do                      ▷  $O(\text{resorte})$  acotado por  $O(\text{dimension})$ 
8:     if validarBordes(dimension, actualX, j) then                  ▷  $O(1)$ 
9:       Nodo alcanzableDir ← new Nodo(actualX + j, actualY, nivelActual) ▷  $O(1)$ 
10:      alcanzables.agregar(alcanzableDir)                            ▷  $O(1)$ 
11:    end if
12:    if validarBordes(dimension, actualY, j) then                  ▷  $O(1)$ 
13:      Nodo alcanzableDir ← new Nodo(actualX, actualY + j, nivelActual) ▷  $O(1)$ 
14:      alcanzables.agregar(alcanzableDir);                            ▷  $O(1)$ 
15:    end if
16:  end for
17:  for int potAdic ← 1; potAdic+ ≤ nivelActual; potAdic ++ do      ▷  $O(\text{dimension})$ 
18:    int potTotal ← potencia + potAdic;
19:    if potTotal > dimension then
20:      break
21:    end if
22:    if validarBordes(dimension, actualY, potTotal) then            ▷  $O(1)$ 
23:      NodoalcanzableIndir = new Nodo(actualX, actualY + potTotal, nivelActual - potAdic)
▷  $O(1)$ 
24:      alcanzables.agregar(alcanzableIndir)                            ▷  $O(1)$ 
25:    end if
26:    if validarBordes(dimension, actualY, -potTotal) then          ▷  $O(1)$ 
27:      NodoalcanzableIndir = new Nodo(actualX, actualY - potTotal, nivelActual - potAdic)
▷  $O(1)$ 
28:      alcanzables.agregar(alcanzableIndir)                            ▷  $O(1)$ 
29:    end if
30:    if validarBordes(dimension, actualX, potTotal) then            ▷  $O(1)$ 
31:      NodoalcanzableIndir = new Nodo(actualX + potTotal, actualY, nivelActual - potAdic)
▷  $O(1)$ 
32:      alcanzables.agregar(alcanzableIndir)                            ▷  $O(1)$ 

```

```

33:      end if
34:      if validarBordes(dimension, actualX,  $-potTotal$ ) then  $\triangleright O(1)$ 
35:          NodoalcanzableIndir = newNodo(actualX - potTotal, actualY, nivelActual - potAdic)
 $\triangleright O(1)$ 
36:          alcanzables.agregar(alcanzableIndir)  $\triangleright O(1)$ 
37:      end if
38:  end for
39:  return alcanzables  $\triangleright O(1)$ 
40: end procedure

```

Donde *nodo* es el nodo del cual se van a calcular sus alcanzables, *valorNodo* la potencia de su resorte y *dimension* es n . Podemos ver que al algoritmo lo componen dos ciclos for lineales, que van armando la lista *alcanzables*. El primer ciclo, calcula los nodos alcanzables *Directos*, es decir, aquellos a los que puede saltar con el resorte de la celda sin hacer uso de ninguna potencia adicional. El ciclo itera de $-resorte$ a $resorte$, agregando de ésta forma a la lista de alcanzables los nodos a su izquierda, derecha, hacia adelante y atrás, validando con cuidado los bordes de la matriz, y sin agregarse a sí mismo, tomando cada uno de estos pasos ($O(1)$). Dado que la potencia de un resorte está acotada por la dimensión de la matriz (por la forma en la que generamos el grafo), este ciclo itera como máximo de $-dimension$ a $dimension$, por lo tanto $\in O(n)$.

El segundo ciclo for calcula todos los nodos alcanzables indirectamente, es decir, aquellos para los cuales el jugador tiene que 1 o más de sus potencias extra para poder llegar. Este ciclo itera según la variable *potAdic*, potencias adicionales, que comienza en 1 hasta el nivel del nodo en cuestión (recordar que el nivel de un nodo representa la cantidad de potencias extra que le quedan a un jugador). Este valor se le suma a la potencia del resorte (línea 18) y si se validan bien los bordes, se agregan estos nodos ($O(1)$), los que están a distancia *potencia de resorte* + *potAdic*, y al iterar *potAdic* hasta el nivel del nodo obtenemos todos los nodos alcanzables usando las potencias que le quedan disponibles al jugador. Estas acciones se realizan en $O(1)$, y el ciclo itera de 1 hasta el nivel del nodo.

Esto indicaría que el ciclo tendría una complejidad de $O(k)$, por ser k el nivel máximo del grafo, pero podemos acotar esto, en la línea 19, por el mismo motivo que en el ciclo anterior, una vez que la potencia total a usar supera el valor de la dimensión de la matriz, cortamos el ciclo, ya que no tiene sentido saltar fuera del campo de juego. Por lo tanto calculamos para los posibles valores de potencia adicional que le queden al jugador, hasta que la potencia total de salto sea de *dimension*. Por lo tanto el ciclo también $\in O(dimension)$. Como comentario adicional, remarcamos que es posible agregar este condicional a la guarda del ciclo, pero dada la complejidad de entendimiento de este pseudocódigo decidimos ponerlo como una instrucción de *break* para mayor claridad.

Concluimos en que la función tiene una complejidad de $O(n) + O(n) = O(n)$. Lo cual tiene sentido, ya que si consideramos la matriz de juego, un nodo como máximo puede moverse hacia todos los de su fila y columna, es decir a $2n - 2$ nodos, lo cual concuerda con la complejidad ya que anadimos cada uno de ellos a la lista de alcanzables en tiempo constante.

3.4.3. Búsqueda del camino mínimo

A continuación el pseudocódigo de la búsqueda BFS. Se trata de una modificación de BFS que busca un camino entre el nodo inicial y alguno de los nodos destino, sin importar a qué nivel pertenezcan, es decir, cuando la búsqueda encuentra un nodo w tal que las primeras 2 coordenadas de w coinciden con la celda destino, termina el algoritmo. Acto seguido reconstruye el camino en base a los predecesores, que se van guardando a medida que se realiza la expansión en anchura. Teniendo en cuenta que BFS encuentra el camino mas corto en cantidad de aristas entre 2 nodos, esto resuelve nuestro problema. La salida es una lista de nodos que comprende el camino, indicando en cada paso: (posX, posY, gasto de PowerUp en el salto del predecesor de este nodo al actual).

```

1: procedure CAMINOMINIMO(Juegoj) List < Nodo >
2:   Cola < Nodo > cola  $\leftarrow$  new Cola < Nodo >  $\triangleright O(1)$ 

```

```

3:  cola.encolar(nodoInicial)                                ▷  $O(1)$ 
4:  marcarVisitado(nodoInicial)                              ▷  $O(1)$ 
5:  while  $\neg$ cola.esVacia() do                                ▷  $O(dimension^2)$ 
6:      Nodo actual  $\leftarrow$  cola.desencolar()                ▷  $O(1)$ 
7:      if esDestino(actual) then                              ▷  $O(1)$ 
8:          return construirCaminoConPredecesores(actual)      ▷  $O(dimension^2 * potInicial)$ 
9:      end if
10:     Lista < Nodo > alcanzables  $\leftarrow$  obtenerAlcanzables(actual) ▷  $O(dimension)$ 
11:     for all Nodo alcanzable : alcanzables do                ▷  $O(dimension)$ 
12:         if  $\neg$ estaVisitado(alcanzable) then                  ▷
13:             marcarVisitado(alcanzable)                      ▷  $O(1)$ 
14:             asignarPredecesor(alcanzable, actual)            ▷  $O(1)$ 
15:             cola.encolar(alcanzable)                          ▷  $O(1)$ 
16:         end if
17:     end for
18: end while
19: return null                                                ▷  $O(1)$ 
20: end procedure

```

Podemos ver que el código es análogo al de el algoritmo BFS, con el agregado de que además de marcar cada nodo i una vez visitado, se le asigna un predecesor j , que es el nodo para el cual i pertenece a la lista de alcanzables de j , en el orden de la búsqueda. De esta forma, cuando encuentra al nodo *destino*, puede reconstruir el camino que realizó la búsqueda visitando todos los predecesores hasta llegar al nodo *origen*, de lo cual se encarga la próxima función.

El while principal del ciclo recorre todos los nodos en el peor caso (cuando la búsqueda no se detiene en la línea 7. En cada iteración, se visita un nodo, se lo marca para no visitarlo nuevamente, y se verifica que no sea el nodo que estamos buscando (nodo destino). Si lo es, termina el algoritmo de búsqueda y se llama a la función *construirCaminoConPredecesores*. Caso contrario, se agregan a la cola todos los nodos adyacentes (alcanzables) no visitados, además se indica para cada uno de ellos que el nodo anterior es su predecesor.

Obtener los adyacentes de un nodo se realiza en tiempo constante $O(1)$ (acceder al hashmap por clave) y $O(n)$ para encolar todos los adyacentes (anteriormente argumentado). Sea $f(G, i)$ = cantidad de nodos sin visitar en G en la iteración i , esta función tiene un valor inicial $f(G, 0) = n^2 * k$ y en cada iteración esta función decrece y toma el valor cero en $f(G, n^2 * k) = f(G, cantNodosG) = 0$. En este momento, no hay más nodos no visitados para agregar a la cola. Luego, se procesarán todos los elementos encolados y terminará el algoritmo.

Juntando todas estas observaciones vemos que el *while(!queue.isEmpty())* está asociado a la función $f(G, i)$, que decrece en cada iteración al menos una unidad, en el peor de los casos, tomara $O(f(G, 0)) = O(n^2 * k) = cantNodos\ de\ G\ al\ comenzar$ en recorrer todos los nodos. La función *esDestino(...)* toma tiempo constante, de ingresar en este if, termina el algoritmo con un costo $O(f(G, 0))$ adicional por armar el camino con la función *construirCaminoConPredecesores*. La lista de alcanzables se obtiene en $O(1)$, iterar sobre los alcanzables realizando operaciones de tiempo constante dentro de este subciclo toma $O(n)$. En total, este método tiene una complejidad temporal: iterar sobre todos los nodos y fijarse si es destino: $O(k * n^2) * (O(1))$, obtener nodos alcanzables: $O(1)$, encolar lcanzables: $O(n)$ y al finalizar armar el camino: $O(k * n^2) = O(k * n^2) * O(n) + O(k * n^2) = O(k * n^3)$.

3.4.4. Construcción del camino

Finalmente, el pseudocódigo de la función que genera el camino a partir del recorrido que realizó el BFS. La salida del algoritmo es del tipo *Lista < Nodo >*.

```

1: procedure CONSTRUIRCAMINOCONPREDECESORES(Nodo actual)
2:     Lista < Nodo > camino  $\leftarrow$  new Lista < Nodo >                                ▷  $O(1)$ 

```

```

3:  Nodo nodo ← actual ▷  $O(1)$ 
4:  while damePredecesor(nodo)! = null do ▷  $O(\text{dimension} * \text{dimension})$ 
5:      Nodo predecesor ← damePredecesor(nodo) ▷  $O(1)$ 
6:      nodo.nivel ← predecesor.nivel() – nodo.nivel() ▷  $O(1)$ 
7:      camino.agregarAdelante(nodo) ▷  $O(1)$ 
8:      nodo ← predecesor ▷  $O(1)$ 
9:  end while
10:  nodo.setLevel(0) ▷  $O(1)$ 
11:  camino.agregarAdelante(nodo) ▷  $O(1)$ 
12:  return camino ▷  $O(1)$ 
13: end procedure

```

La lista de nodos del camino va a ser a lo sumo n^2 , dandonos una complejidad de $O(n^2)$ dado que el ciclo que arma el camino itera una vez por cada nodo en base a los predecesores. Adentro del ciclo se realizan operaciones de tiempo constante, dandonos una complejidad temporal total de $O(n^2)$.

Dado que el algoritmo BFS nos brinda el camino más corto en aristas de el nodo origen al destino, llegamos a la conclusión de que la complejidad depende de n^2 , que es la cantidad de nodos, ya que como máximo va a visitar cada uno de los k nodos $(i, j, 0..k)$ una única vez. Supongamos que esto no ocurre, y el camino mínimo visita al menos dos nodos (i, j, k_1) y (i, j, k_2) . Si volvemos a pensar el grafo como el tablero de juego bidimensional, estaría volviendo a la misma celda otra vez, sólo que con menos unidades de potencia (de ahí la diferenciación de nivel), formando un ciclo, el cual podríamos retirar para quedarnos con un camino más corto. Esto es absurdo, ya que consideramos mínimo el anterior camino, por lo tanto, el camino mínimo va a visitar como máximo una vez cada celda, la complejidad va a ser $O(n^2)$.

3.4.5. Complejidad Total

Complejidad total:

1. Armar el grafo: $O(k * n^3)$
2. Algoritmo de búsqueda en anchura y armado del camino: $O(k * n^3)$

Por lo tanto la complejidad es $O(k * n^3)$ como lo pide el enunciado.

Documentación de la clase HashMap:

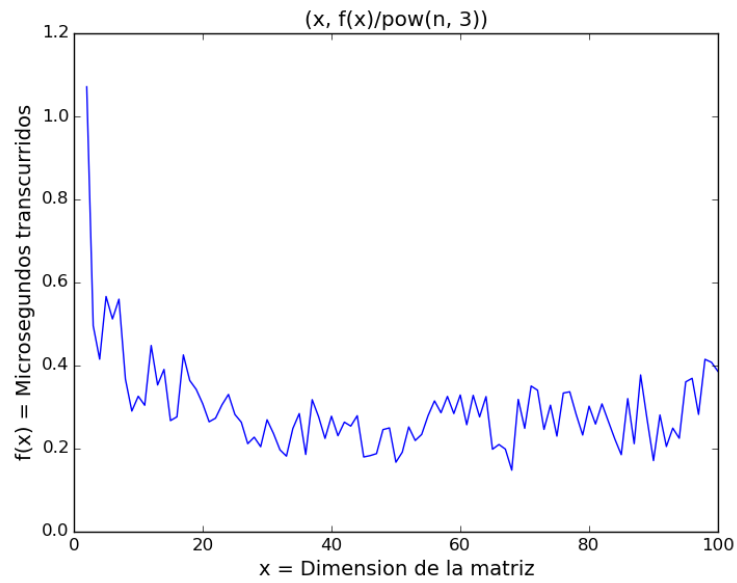
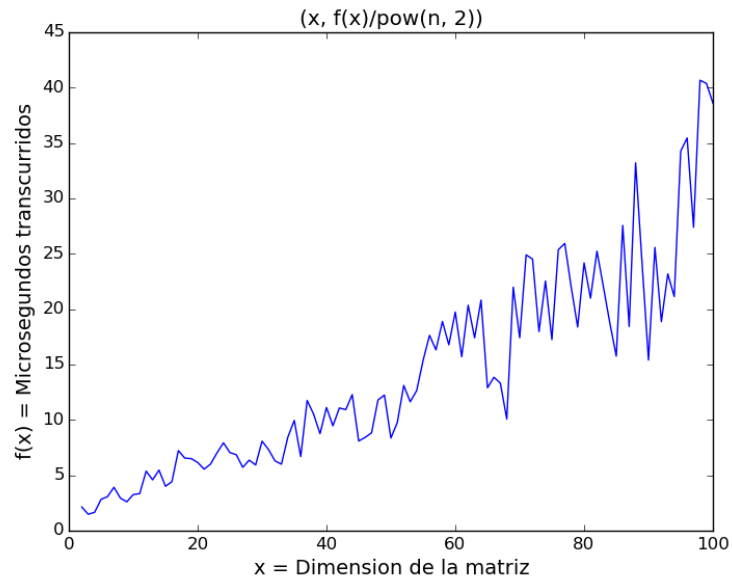
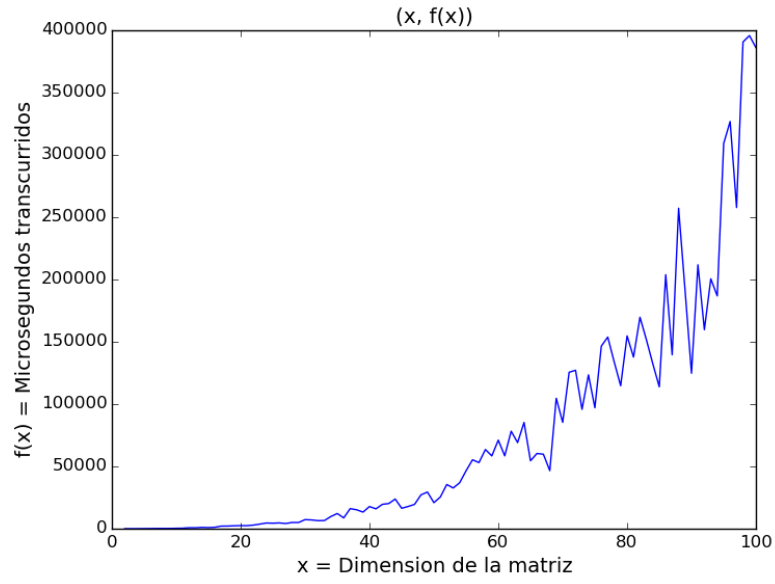
<http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html>

3.5. Mediciones de performance

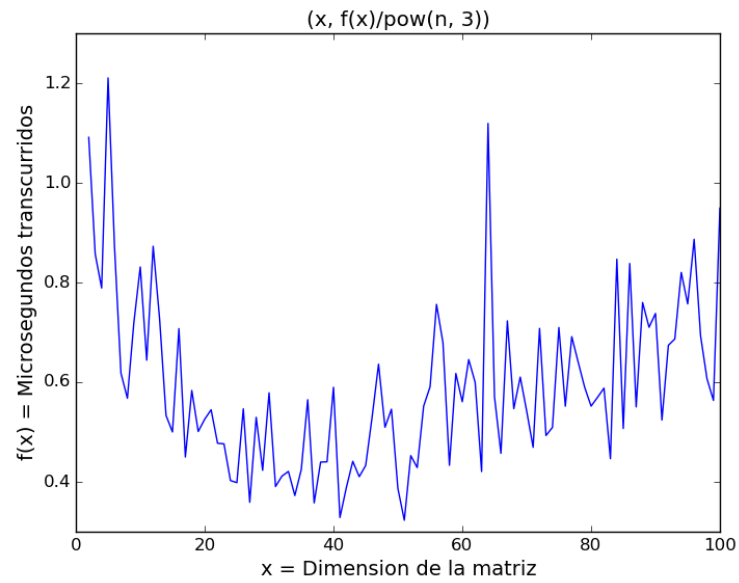
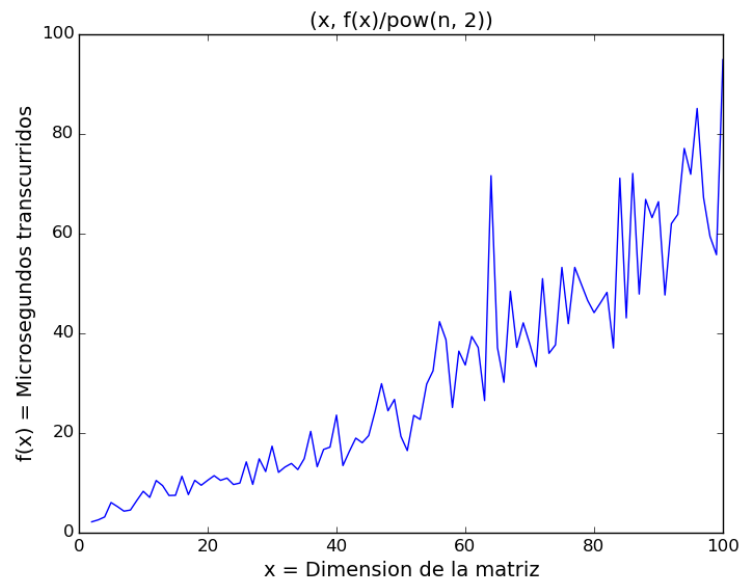
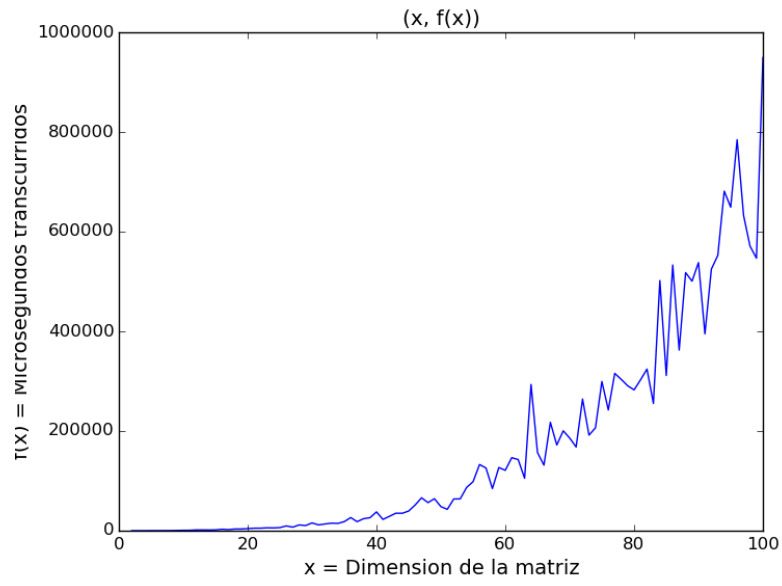
Compilador GJC: Java realiza un profiling sobre el programa a compilar y toma estadísticas para realizar diferentes procesos de compilación/interpretación sobre el código, esto se ve reflejado en el tiempo de ejecución del programa compilado con el jdk, para forzar la compilación de java a código objeto, utilizamos el compilador GCJ de GNU, que es uno de los compiladores denominados Ahead-Of-Time. Todos los experimentos fueron ejecutados sobre el programa compilado a código nativo. La cantidad de iteraciones por cada entrada es 100 para licuar los posibles outliers.

Como $f(n, k) = \{\text{tiempo en milisegundos transcurrido en la ejecución de una entrada de dimension } n \text{ y powerUp } k\}$ es una función que va de $\mathbb{R}^2 \rightarrow \mathbb{R}$, en lugar de graficarla en 3d, graficaremos ciertas curvas $C_k = \{(x, f(x, k)) \in \mathbb{R}^2\}$ para ver como se comporta la función cuando $n \rightarrow \infty$ con k fijo

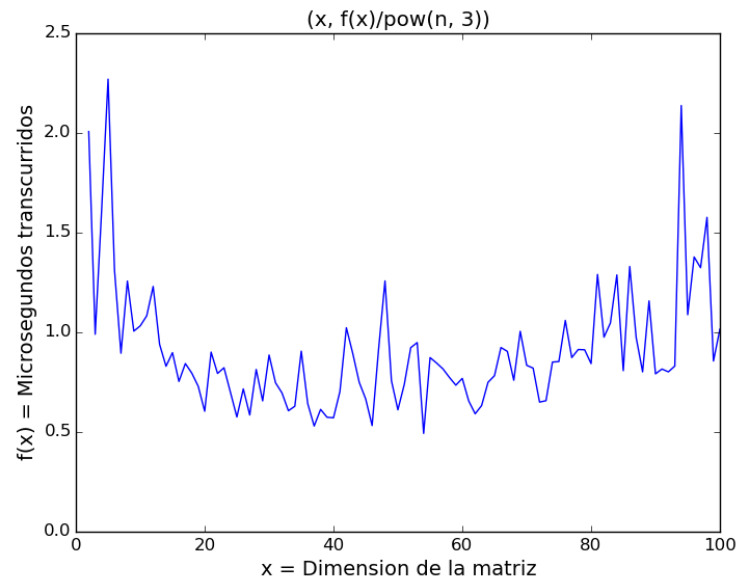
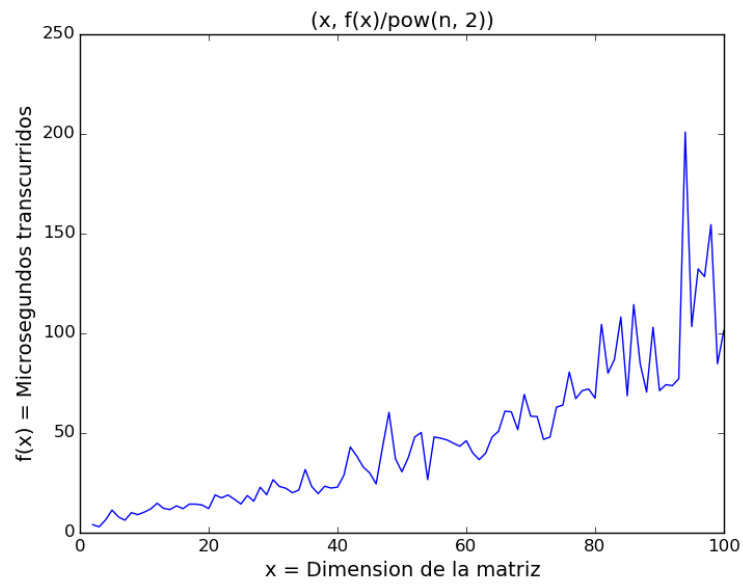
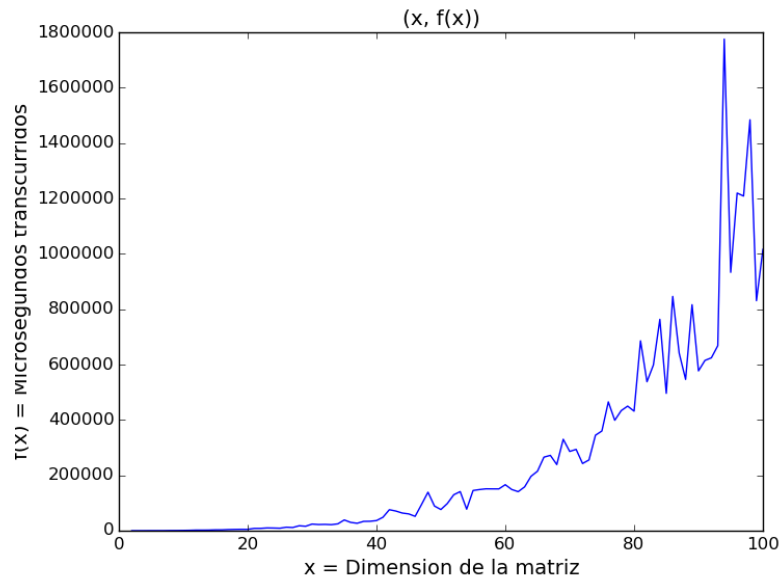
- $C_0 = \{(x, f(x, 0)) \in \mathbb{R}^2\}$



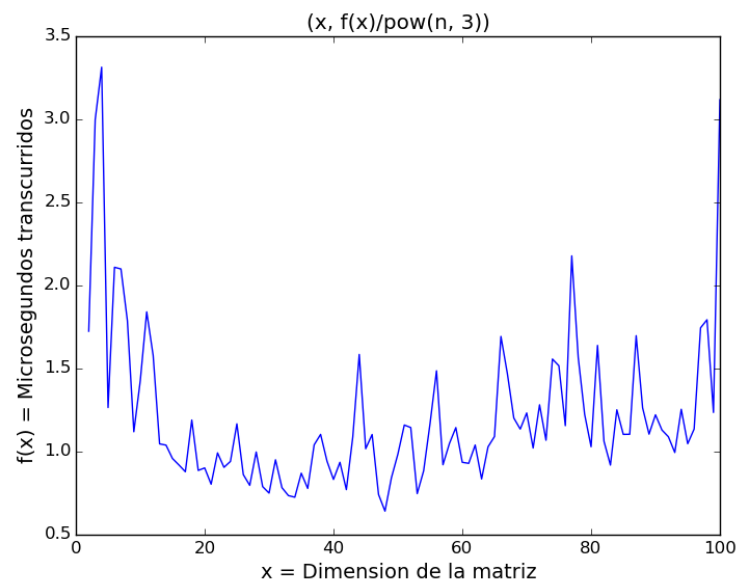
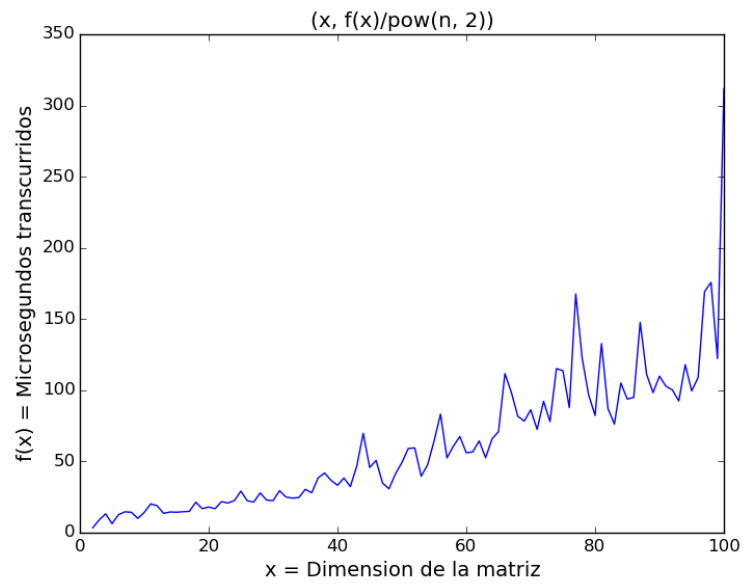
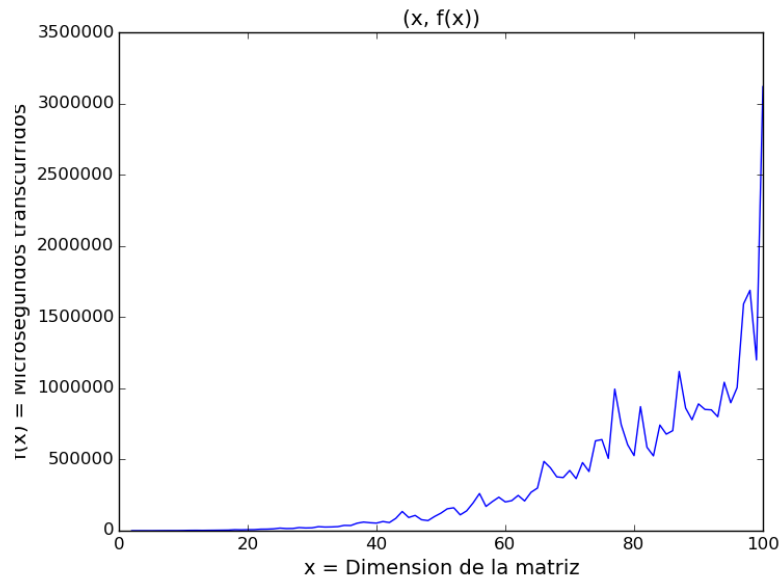
■ $C_1 = \{(x, f(x, 1)) \in \mathbb{R}^2\}$



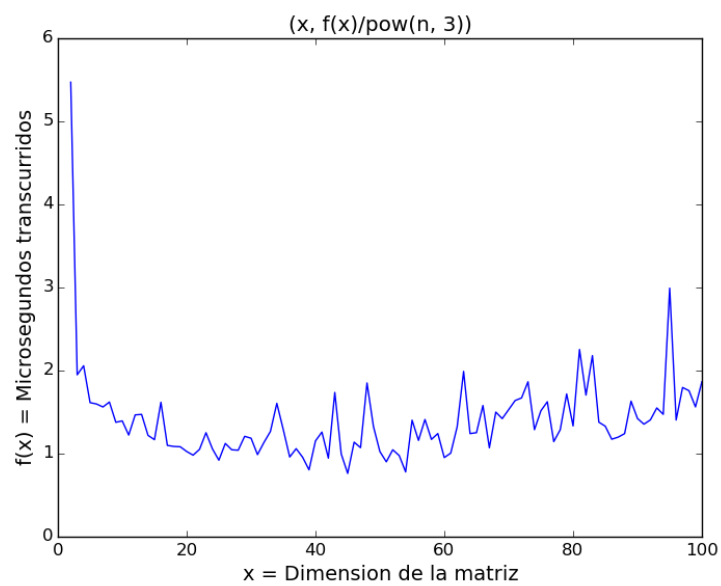
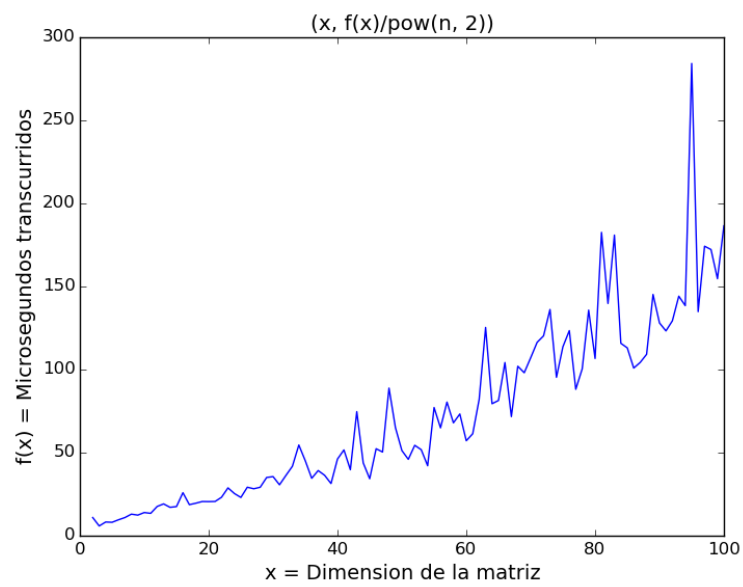
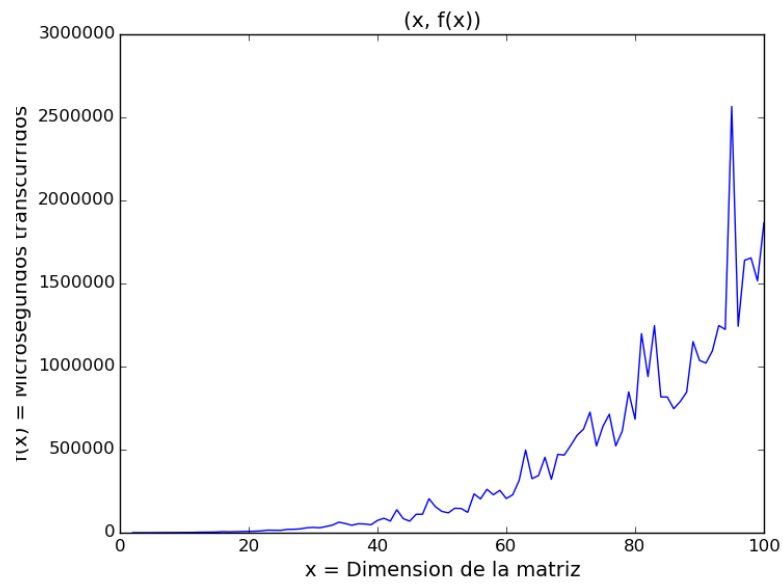
■ $C_2 = \{(x, f(x, 2)) \in \mathbb{R}^2\}$



■ $C_3 = \{(x, f(x, 3)) \in \mathbb{R}^2\}$



■ $C_4 = \{(x, f(x, 4)) \in \mathbb{R}^2\}$



Interpretacion de los graficos: Como se puede ver en los graficos, la complejidad empirica, coincide

con la complejidad teorica esperada $O(k * n^3)$, llegamos a esta conclusion graficando los siguientes casos: $k = [0.,5]$ $n = [2.,100]$ para cada caso, se realizaron 3 graficos, $(X, F(X))$, $(X, F(X)/x^2)$ y $(X, F(X)/x^3)$. En estos graficos se puede observar que al dividir por x^2 se obtiene una curva con crecimiento lineal aproximado, y al dividir por x^3 , se obtiene una constante aproximada, es decir, una curva acotada entre una diferencia pequena en el eje X, concluyendo que la curva inicial corresponde a una funcion $f(n) = q.n^3$, con q en \mathbb{R} . Asimismo veamos que cuando se incrementa el valor de k para los graficos anteriores, la imagen de las funciones graficadas crecen linealmente a medida que crece k . Lo que nos indica que en realidad las funciones graficadas son realmente $f(n) = j.(k.n^3)$, con j en \mathbb{R} , concluyendo, $f(n) \in O(k * n^3)$. Referencia: <http://www.excelsior-usa.com/jetcs00007.html>

4. Apéndice: Generación de casos de prueba

Para el testeo de los algoritmos y la medición de tiempos en función de la entrada, se programó una utilidad para generar los casos de prueba.

Los números aleatorios que se generaron en los casos, se hizo con la función *random()* de C (<http://linux.die.net/man/3/random>) usando como semilla el tiempo en microsegundos.

Para el ejercicio 2, se le indica al programa la cantidad de ciudades, la cual se fue incrementando para la medición del tiempo, y la cantidad de centrales se indicaba como un número aleatorio entre 1 y la cantidad de ciudades.

Así se crearon muchos casos donde se fue incrementando el tamaño de la entrada y ejecutando varios tests de un mismo tamaño y varias veces el mismo test para obtener un promedio del tiempo que tarda en resolverlo y descartar algunas imperfecciones que puedan surgir por el entorno donde se está midiendo los tiempos, como puede ser que justo se ejecute otra tarea.

Para los ejercicios 1 y 3, se programaron dentro de las clases Main de los diferentes ejercicios diferentes metodos que se encargan de la codificación y decodificación de entrada y salida tanto como de la generación de tests aleatorios con distribución uniforme. Se indica por parametro al ejecutable que se desea realizar.

5. Apéndice: Código fuente relevante

5.1. Ejercicio 1

```
1 public class RobaNumeros {
2
3     private int turnosJugados;
4     private int puntosJugador1;
5     private int puntosJugador2;
6     private TuplaTurnos turnos;
7
8     public void calcular( int[] entrada ){
9         //Calcula las tablas S y P
10        Tablas_S_P tabla_res = new Tablas_S_P(entrada);
11
12        //Las usa para obtener los datos requeridos por el enunciado
13
14        turnos = new TuplaTurnos(entrada.length); //Como maximo se juegan tantos turnos como
15            cantidad de cartas
16        turnosJugados=1; //se juega siempre al menos un turno, el primero
17        puntosJugador1 = 0;
18        puntosJugador2 = 0;
19        int i = 1;
20        int j = entrada.length;
21        boolean jugador = true; //true es jugador1
22        while(tabla_res.P()[i-1][j-1][0] != 0){ //La condicion es true si no se agarraron todas
23            las cartas
24
25            //calcula lado
26            if(tabla_res.P()[i-1][j-1][0] != i){
27                turnos.lado[turnosJugados-1] = true;
28            }
29            else{
30                turnos.lado[turnosJugados-1] = false;
31            }
32            //calcula cant de cartas robadas
33            turnos.cant[turnosJugados-1] = (j - i) - (tabla_res.P()[i-1][j-1][1] - tabla_res.P
34                () [i-1][j-1][0]);
35
36            //sumo puntos para el jugador que juega actualmente
37            if(turnos.lado[turnosJugados-1] == true){ //Si se saco de izquierda
38                if(jugador == true){
39                    puntosJugador1 += tabla_res.S()[ i-1 ][ i + turnos.cant[turnosJugados-1]
40                        -2];
41                }
42                else{
43                    puntosJugador2 += tabla_res.S()[ i-1 ][ i + turnos.cant[turnosJugados-1]
44                        -2];
45                }
46            }
47            else{ //Si se saco de derecha
48                if(jugador == true){
49                    puntosJugador1 += tabla_res.S()[ j - (turnos.cant[turnosJugados-1] -1) -1][
50                        j-1];
51                }
52                else{
53                    puntosJugador2 += tabla_res.S()[ j - (turnos.cant[turnosJugados-1] -1) -1][
54                        j-1];
55                }
56            }
57
58            int i2 = i;
59            i = tabla_res.P()[i-1][j-1][0];
60            j = tabla_res.P()[i2-1][j-1][1];
61            jugador = !jugador;
62            turnosJugados++;
63        }
64
65        //Falta lo correspondiente al ultimo turno
66        turnos.lado[turnosJugados-1] = true; // Se elige lado por defecto "izq" para cuando se
67            sacan todas
68        turnos.cant[turnosJugados-1] = j - i + 1;
69        if(jugador == true){
70            puntosJugador1 += tabla_res.S()[i-1][j-1];
71        }
72        else{
73            puntosJugador2 += tabla_res.S()[i-1][j-1];
74        }
75    }
76}
```

```

65         puntosJugador2 += tabla_res.S()[i-1][j-1];
66     }
67
68 }
69
70 public int getTurnosJugados(){
71     return turnosJugados;
72 }
73
74 public int getPuntosJugador1(){
75     return puntosJugador1;
76 }
77
78 public int getPuntosJugador2(){
79     return puntosJugador2;
80 }
81
82 public boolean esTurnoIzquierdo(int turno){
83     return turnos.lado[turno-1];
84 }
85
86 public int getCantidadRobadaPorTurno(int turno){
87     return turnos.cant[turno-1];
88 }
89
90 private class TuplaTurnos{
91
92     public TuplaTurnos(int cant_turnos){
93         lado = new boolean[cant_turnos];
94         cant = new int[cant_turnos];
95     }
96
97     public boolean lado[]; //true es "izq"
98     public int cant[];
99
100 }
101
102 private class Tablas_S_P {
103
104     private int S[][];
105     private int P[][][]; // P[i][j] es una tupla que en las dos primeras coordenadas tiene
106                             el i y el j que quedan en la mesa despues de jugado el turno ( $i, \frac{1}{2}$  (0,0) si se
107                             levantan todas las cartas) y en la tercer coordenada tiene a la funcion P
108                             propiamente.
109     private int tira_entrada[];
110
111     public Tablas_S_P(int entrada[]){
112         tira_entrada = entrada;
113         S = new int[tira_entrada.length][tira_entrada.length];
114         //Calcula tabla S
115         calcular_S();
116         P = new int[tira_entrada.length][tira_entrada.length][3];
117         //Calcula tabla P a partir de la S
118         calcular_P();
119     }
120
121     public int[][][] P(){
122         return P;
123     }
124
125     public int[][] S(){
126         return S;
127     }
128
129     private void calcular_S(){
130         int i;
131         for(i=1;i<=tira_entrada.length;i++){
132             int j;
133             for(j=i;j<=tira_entrada.length;j++){
134                 if(i==j){
135                     S[i-1][j-1] = tira_entrada[i-1];
136                 }
137                 else{
138                     S[i-1][j-1] = S[i-1][j-2] + tira_entrada[j-1];
139                 }
140             }
141         }
142     }
143 }

```



```

139
140     private void calcular_P() { //Recorre la tabla de columna 1 hacia columna n, cada una de
141         abajo hacia arriba.
142         int j;
143         for (j=1; j<=tira_entrada.length;j++){
144             int i;
145             for (i=j; i>=1;i--){
146
147                 int [] restante = {i,j};
148                 Double MAX = Double.NEGATIVE_INFINITY;
149                 int k;
150                 //Para cada posicion, que representa un intervalo (i,j) de cartas,
151                 //calcula la eleccion de cartas a robar, primero desde la izquierda y
152                 //luego desde la derecha
153                 //En cada posicion queda guardada una tupla con el valor de P y ademas
154                 //los indices (i,j) del intervalo de cartas que quedan por jugarse,
155                 //i.e.  $\frac{1}{2}$  (0,0) en caso robar todo.
156                 for (k=0;k<=j-i;k++){
157                     int m;
158                     m = S[i-1][i+k-1] - ( (k==j-i)?0:P[i+k][j-1][2] );
159                     if (m > MAX) {
160                         MAX = (double) m;
161                         if (k==j-i) {
162                             restante[0]=0;
163                             restante[1]=0;
164                         }
165                         else {
166                             restante[0]=i+k+1;
167                             restante[1]=j;
168                         }
169                     }
170                 }
171                 for (k=0;k<=j-i;k++){
172                     int m;
173                     m = S[j-k-1][j-1] - ( (k==j-i)?0:P[i-1][j-k-2][2] );
174                     if (m > MAX) {
175                         MAX = (double) m;
176                         if (k==j-i) {
177                             restante[0]=0;
178                             restante[1]=0;
179                         }
180                         else {
181                             restante[0]=i;
182                             restante[1]=j-k-1;
183                         }
184                     }
185                 }
186                 double _MAX = MAX;
187                 int tupla [] = {restante[0], restante[1], (int)_MAX};
188                 P[i-1][j-1] = tupla;
189             }
190         }
191     }
192 }
193

```

5.2. Ejercicio 2

```

1  typedef int  Nodo;
2
3  struct GrafoMatrizAdyacencia{
4      int  nodos; //cantidad de nodos
5      int  *componente.conexa_nodos; //array para asignar componentes conexas
6      int  componentes.conexas;
7      int  **adyacencia; //la matriz ed adyacencia
8      int  aristas; //aristas colocadas
9  };
10
11  typedef struct GrafoMatrizAdyacencia Grafo;
12
13  typedef struct Ciudad {
14      Nodo nodo;

```

```

15     int x;
16     int y;
17 } Ciudad;
18
19 typedef struct NodoDistancia {
20     int agregado;
21     int distancia;
22     Nodo nodo;
23 } NodoDistancia;
24
25 typedef struct Arista{
26     Nodo nodo1;
27     Nodo nodo2;
28     int distancia;
29 } Arista;
30
31 Grafo *crear_grafo(int nodos)
32 {
33     int i;
34
35     Grafo *g = NULL;
36     if(nodos <= 0)
37         return NULL;
38
39     g = (Grafo *)malloc(sizeof(Grafo));
40     g->nodos = nodos;
41     g->componente_conexa_nodos = (int *)calloc(nodos, sizeof(int));
42     g->componentes_conexas = nodos;
43     g->aristas = 0;
44     g->adyacencia = (int **)malloc(sizeof(int *) * nodos);
45
46     for(i = 0; i < nodos; i++) {
47         g->adyacencia[i] = (int *)calloc(nodos, sizeof(int));
48     }
49     for(i = 0; i < nodos; i++) {
50         g->componente_conexa_nodos[i] = i;
51     }
52     return g;
53 }
54
55 void liberar_grafo(Grafo *g)
56 {
57     int i;
58
59     if(!g)
60         return;
61
62     free(g->componente_conexa_nodos);
63     for(i = 0; i < g->nodos; i++) {
64         free(g->adyacencia[i]);
65     }
66     free(g->adyacencia);
67     free(g);
68 }
69
70 int agregar_arista(Grafo *g, Nodo nodo1, Nodo nodo2)
71 {
72     int i;
73     int nueva_componente, vieja_componente;
74
75     if(!g)
76         return -1;
77
78     if(nodo1 >= g->nodos || nodo2 >= g->nodos)
79         return -1;
80
81     g->adyacencia[nodo1][nodo2] = 1;
82     g->adyacencia[nodo2][nodo1] = 1;
83     g->aristas++;
84
85     if(g->componente_conexa_nodos[nodo1] != g->componente_conexa_nodos[nodo2]){
86         nueva_componente = g->componente_conexa_nodos[nodo1];
87         vieja_componente = g->componente_conexa_nodos[nodo2];
88
89         for(i = 0; i < g->nodos; i++){ //actualizo la componente conexa de todos los nodos que
90             pertenecian a esa componente
91             if(g->componente_conexa_nodos[i] == vieja_componente){

```

```

91         g->componente_conexa_nodos[i] = nueva_componente;
92     }
93 }
94 g->componentes_conexas--;
95 }
96 return 0;
97 }
98
99 int cantidad_componentes_conexas(Grafo *g)
100 {
101     if(!g)
102         return -1;
103     return g->componentes_conexas;
104 }
105
106 int cantidad_aristas(Grafo *g)
107 {
108     if(!g)
109         return -1;
110     return g->aristas;
111 }
112
113 int cantidad_nodos(Grafo *g)
114 {
115     if(!g)
116         return -1;
117     return g->nodos;
118 }
119
120 int son_adyacentes(Grafo *g, Nodo nodo1, Nodo nodo2)
121 {
122     if(!g)
123         return 0;
124
125     return g->adyacencia[nodo1][nodo2];
126 }
127
128 //retorna un vector con cantidad_componentes_conexas() elementos, en cada iesimo elemento, hay
129 //un nodo correspondiente a la componente iesima. liberar el resultado con free()
130 Nodo *nodos_de_componentes(Grafo *g)
131 {
132     Nodo *nodos;
133     int i;
134
135     if(!g)
136         return NULL;
137
138     nodos = (Nodo *)malloc(sizeof(Nodo) * g->nodos);
139     for(i = 0; i < g->nodos; i++){
140         nodos[i] = -1;
141     }
142     for(i = 0; i < g->nodos; i++){
143         nodos[g->componente_conexa_nodos[i]] = i;
144     }
145     return nodos;
146 }
147
148 int distancia(Ciudad *c1, Ciudad *c2)
149 {
150     if(!c1 || !c2)
151         return 0;
152
153     return (c1->x - c2->x) * (c1->x - c2->x) + (c1->y - c2->y) * (c1->y - c2->y); //el cuadrado
154     //de la distancia euclidiana
155 }
156
157 void ordenar_aristas(Arista *aristas, int n)
158 {
159     int m, i, j, k;
160     Arista *aux;
161
162     if(n <= 1)
163         return;
164     m = n / 2;
165     ordenar_aristas(aristas, m);
166     ordenar_aristas(aristas + m, n - m);
167     aux = (Arista *)malloc(sizeof(Arista) * n);

```

```

166 i = 0;
167 j = 0;
168 k = 0;
169 while(i < m || j < (n - m)){
170     if(j >= (n - m)){
171         memcpy(&(aux[k]), &(aristas[i]), sizeof(Arista));
172         i++;
173         k++;
174         continue;
175     }
176     if(i >= m){
177         memcpy(&(aux[k]), &(aristas[m + j]), sizeof(Arista));
178         j++;
179         k++;
180         continue;
181     }
182     if(aristas[i].distancia < aristas[m + j].distancia){
183         memcpy(&(aux[k]), &(aristas[i]), sizeof(Arista));
184         i++;
185         k++;
186         continue;
187     }
188     else{
189         memcpy(&(aux[k]), &(aristas[m + j]), sizeof(Arista));
190         j++;
191         k++;
192         continue;
193     }
194 }
195 memcpy(aristas, aux, n * sizeof(Arista));
196 free(aux);
197 }
198
199 Grafo *resolver(int k-centrales, Ciudad *ciudades, int n-ciudades, Nodo **centrales)
200 {
201     Grafo *g = NULL;
202     NodoDistancia *nodos = NULL;
203     Arista *aristas = NULL;
204     int i, agregados, distancia_minima = -1, nodo_minimo = -1, componentes;
205
206     if(k-centrales <= 0 || ciudades == NULL || n-ciudades <= 0 || centrales == NULL){
207         return NULL;
208     }
209
210     g = crear_grafo(n-ciudades);
211     nodos = (NodoDistancia *)malloc(sizeof(NodoDistancia) * n-ciudades);
212     aristas = (Arista *)malloc(sizeof(Arista) * (n-ciudades - 1));
213
214     nodos[0].agregado = 1;
215     nodos[0].distancia = 0;
216     nodos[0].nodo = 0;
217     for(i = 1; i < n-ciudades; i++){
218         nodos[i].agregado = 0;
219         nodos[i].distancia = distancia(&(ciudades[i]), &(ciudades[0]));
220         nodos[i].nodo = 0;
221     }
222
223     for(agregados = 0; agregados < n-ciudades - 1; agregados++){
224         distancia_minima = -1;
225         nodo_minimo = 0;
226         for(i = 0; i < n-ciudades; i++){
227             if(!nodos[i].agregado){
228                 if(distancia_minima == -1 || nodos[i].distancia < distancia_minima){
229                     distancia_minima = nodos[i].distancia;
230                     nodo_minimo = i;
231                 }
232             }
233         }
234
235         nodos[nodo_minimo].agregado = 1;
236         aristas[agregados].nodo1 = nodo_minimo;
237         aristas[agregados].nodo2 = nodos[nodo_minimo].nodo;
238         aristas[agregados].distancia = distancia_minima;
239
240         for(i = 0; i < n-ciudades; i++){
241             if(!nodos[i].agregado){
242                 if(nodos[i].distancia > distancia(&(ciudades[i]), &(ciudades[nodo_minimo]))){

```

```

243         nodos[i].distancia = distancia(&(ciudades[i]), &(ciudades[nodo_minimo]));
244         nodos[i].nodo = nodo_minimo;
245     }
246 }
247 }
248 }
249
250 ordenar_aristas(aristas, n_ciudades - 1);
251
252 for(componentes = n_ciudades; componentes > k_centrales; componentes--){
253     agregar_arista(g, aristas[n_ciudades - componentes].nodo1, aristas[n_ciudades -
254     componentes].nodo2);
255 }
256
257 *centrales = nodos_de_componentes(g);
258
259 free(nodos);
260 free(aristas);
261
262 return g;
263 }

```

5.3. Ejercicio 3

```

1
2 public class Nodo {
3     //posicion x en el grafo
4     private int x;
5
6     //posicion y en el grafo
7     private int y;
8
9     //posicion z en el grafo
10    private int level;
11
12    //...constructor, getters, setters, toString, hashCode, equals, etc...
13 }
14
15
16 public class NodoMetadata {
17     //predecesor en el grafo
18     private Nodo predecesor;
19
20     // potencia del resorte
21     private int nodoValue;
22
23     // flag para bfs
24     private boolean fueVisitado;
25
26     // Lista de nodos adyacentes
27     private List<Nodo> alcanzables;
28
29     //...constructor, getters, setters, etc...
30 }
31
32 /**
33  *
34  * @author svilerino
35  * Implementacion:
36  * El algoritmo consta de varias etapas principales
37  * 1- Generacion del grafo a partir de los datos de entrada(Constructor de la clase Juego)
38  * 2- Busqueda del camino minimo(en cantidad de pasos) con un BFS.(metodo caminoMinimo)
39  * 3- Construccion del camino a travÃ©s de los predecesores del destino indicados por BFS
40  * y calculo de powerup utilizado en cada salto. (metodo construirCaminoConPredecesores)
41  *
42  * Complejidad:
43  * 1- Los 3 for anidados indican una complejidad de  $O(k \cdot \text{pow}(n, 2))$ , adentro de estos ciclos
44  * se realizan operaciones, todas toman  $O(1)$  salvo la llamada a alcanzables(...) de la
45  * que hablaremos
46  * mas abajo, pero toma  $O(n)$ , dandonos un total de  $O(k \cdot \text{pow}(n, 3))$  en la construccion del
47  * grafo para nuestro modelo.
48  * En el informe se detallara mas acerca de esto, pero veamos que cada nodo puede
49  * tener a lo sumo  $2n$  alcanzables.
50  * Asumiendo que acotamos  $\text{potencia}(v) \leq \text{dimensionMatriz}$  de aqui en adelante tenemos:
51  * Dado que podemos particionar  $\text{alcanzables}(v) = \{\text{alcanzables con potTotal} \leq \text{potencia}(v)\}$ 

```

```

49 *      U disj {alcanzables con potTotal = potencia(v) + powerUpK1} U disj {alcanzables con
    potTotal = potencia(v) + powerUpK2}
50 *      ...U disj {alcanzables con potTotal = potencia(v) + powerUpKn}, en el mejor de los
    casos para este analisis, toda la fila y columna
51 *      de un nodo seran alcanzables sin utilizar powerup, lo cual nos dara el primer
    conjunto union conjuntos vacios, el cardinal de
52 *      alcanzables(v) es entonces 2n, en otros casos, veamos que si no es alcanzado por la
    potencia intrinseca de la torre, todos los
53 *      conjuntos que se unen tienen distinta potencia total e incluyen a los nodos que son
    alcanzables estrictamente con esa potenciaTotal.
54 *      En el mejor de los casos, de nuevo, usando powerup, entre todos los conjuntos, se
    pueden alcanzar 2n nodos, con lo cual
55 *      viendo los ciclos en el metodo calcularAlcanzables y asumiendo que la operacion put
    de hashmap es O(1) amortizada, nos da
56 *      una complejidad total de O(n).
57 *
58 *      2- Sea n = {dimension de la matriz cuadrada de entrada}, k = unidades de powerUp
    Disponibles al inicio del algoritmo + 1(hay capas [0..k] en el grafo)
59 *      El while principal del ciclo recorre todos los nodos en el peor caso(cuando la
    busqueda no se detiene en el if(esDestino(...))
60 *      Veamos porque: En cada iteracion, se visita un nodo, se lo marca para no visitarlo
    nuevamente, y se verifica que no sea
61 *      el nodo que estamos buscando(nodo destino), si lo es, termina el algoritmo de
    busqueda y se llama a la funcion construirCaminoConPredecesores
62 *      Caso contrario, se agregan a la cola todos los nodos adyacentes(alcanzables) NO
    visitados, ademas se indicara quien es el predecesor, en este paso.
63 *      Veamos ciertos detalles tecnicos, obtener los adyacentes de un nodo es O(1)
    amortizado(acceder al hashmap por key) y O(n) para encolar todos los adyacentes
64 *      (cada nodo tenia 2n alcanzables como maximo). Sea f(G, i) = {cantidad de nodos sin
    visitar en G en la iteracion i}, esta funcion tiene un valor inicial f(G, 0) = pow(n,2)*k
    y
65 *      en cada iteracion esta funcion decrece y toma el valor cero en f(G, pow(n,2)*k) = f
    (G, cantNodosG) = 0, en este momento, no hay mas nodos no visitados para agregar
66 *      a la cola. Luego, se procesaran todos los elementos encolados y terminara el
    algoritmo. Juntando todas estas observaciones vemos que el
67 *      while(!queue.isEmpty()) esta asociado a la funcion f(G, i), que decrece en cada
    iteracion al menos una unidad, en el peor de los casos,
68 *      tomara O(f(G,0)) = O(pow(n,2)*k) = {cantNodos de G al comenzar} en recorrer todos
    los nodos.
69 *      la funcion esDestino(...) toma tiempo constante, de ingresar en este if, termina el
    algoritmo con un costo O(f(G, 0)) adicional por armar
70 *      el camino con la funcion construirCaminoConPredecesores. La lista de alcanzables se
    obtenia en O(1), iterar sobre los alcanzables(adyacentes)
71 *      realizando operaciones de tiempo constante adentro de este subciclo era O(n). En
    total, este metodo tiene una complejidad temporal
72 *      {iterar sobre todos los nodos O(k*pow(n, 2))}*({esDestino O(1)} + {
    obtenerAlcanzables O(1) + {encolarAlcanzables O(n)} + {al finalizar armar camino O(k*pow(n
    , 2))})
73 *      = O(k*pow(n, 2))*O(n) + O(k*pow(n, 2)) = O(k*pow(n, 3))
74 *
75 *      3- Asumiendo que BFS funciona correctamente, la lista de nodos del camino sera a lo
    sumo pow(n,2) (todos los nodos), dandonos
76 *      una complejidad de O(pow(n,2)) dado el while que arma el camino en base a los
    predecesores. Adentro del ciclo se realizan
77 *      operaciones de tiempo constante, dandonos una complejidad temporal total de O(pow(n
    , 2))
78 *
79 *      Complejidad total: {1- armar el grafo O(k*pow(n, 3))} + {2,3- Algoritmo de busqueda en
    anchura con armado de camino O(k*pow(n, 3))} = O(k*pow(n, 3))
80 *
81 *      Referencias de estructuras utilizadas:
82 *      http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html
83 *      ...This implementation provides constant-time performance for the basic operations
    (get and put)...
84 */
85 public class Juego {
86
87     //Nodo inicial
88     Nodo nodoInicial;
89
90     // Nodo destino
91     Nodo nodoDestino;
92
93     //Dada la matriz de valores de cada nodo (i,j) y el powerUpInicial
94     //Podemos definir el grafo(una matriz tridimensional de nodos) (i,j,k) donde 0<=k<=
    powerUpInicial
95     //Defino el grafo como un HashMap<Nodo, NodoMetadata> graph

```

```

96 //que entre otras cosas almacena los nodos adyacentes para cada nodo
97 //El grafo G = <V, E> puede obtenerse como
98 //V = HashMap.keySet() && E = Union(graph.get(V.each()).getAlcanzables())
99 HashMap<Nodo, NodoMetadata> graph;
100
101 /**
102  * Precalculo todos los adyacentes de todos los nodos y me armo el grafo para luego aplicar
103   * el algoritmo que resuelve el problema
104  *
105  * Pre: (valoresNodos tiene dimension n*n con n natural > 0) && (0<=powerUpInicial) && (
106   * valoresNodos[i][j] = valorNodoIJ)
107  * && (nodoInicial y nodoDestino tienen coord validas)
108  * @param valoresNodos
109  * @param powerUpInicial
110  */
111 public Juego(int [][] valoresNodos, int powerUpInicial, int filaInicial, int columnaInicial
112 , int filaDestino, int columnaDestino){
113     //inicializo estructuras y atributos
114     this.nodoInicial = new Nodo(filaInicial, columnaInicial, powerUpInicial);
115     this.nodoDestino = new Nodo(filaDestino, columnaDestino, 0/*dummy level 1, 1 puede ser
116     cualquiera entre [0..powerUpInicial]*/);
117     this.graph = new HashMap<Nodo, NodoMetadata>();
118
119     //genero el grafo que modela el problema
120     int dimension = valoresNodos.length;
121     for (int level=powerUpInicial; level>=0; level--){
122         for (int i=0; i<dimension; i++){
123             for (int j=0; j<dimension; j++){
124                 Nodo nodo = new Nodo(i, j, level);
125                 //de nuevo fix para complejidad, no tiene sentido una potencia intrinseca
126                 //mayor a dim
127                 //nunca va a saltar mas alla de los limites de la matriz
128                 int nodoValue = (valoresNodos[i][j] < dimension) ? valoresNodos[i][j] :
129                 dimension;
130                 boolean fueVisitado = false;
131                 List<Nodo> alcanzables = calcularAlcanzables(nodo, nodoValue, dimension);
132                 NodoMetadata metaData = new NodoMetadata(nodoValue, fueVisitado,
133                 alcanzables, null);
134                 graph.put(nodo, metaData);
135             }
136         }
137     }
138 }
139
140 /**
141  * Sea un nodo v(i,j,k) sus alcanzables directos son
142  * los nodos w de la forma (a, b, k) / ((i-c <= a <= i+c) && (b==j)) || ((j-c <= b <= j+c)
143   * && (a==i)) && c = potencia(v)
144  * ahora , sea l una variable que mueve en el rango [1..k]
145  * los nodos t de la forma (a, b, l) / ((a = i-c) || (a = i+c) && (b==j)) || ((b = j-c) ||
146   * (b = j+c) && (a==i)) && c = potencia(v) + 1
147  * @param nodo
148  * @param nodoValue
149  * @return
150  */
151 private List<Nodo> calcularAlcanzables(Nodo nodo, int nodoValue, int dimensionMatriz) {
152     int currentLevel = nodo.getLevel();
153     int currentX = nodo.getX();
154     int currentY = nodo.getY();
155     int potenciaIntrinseca = nodoValue;
156     LinkedList<Nodo> alcanzables = new LinkedList<Nodo>();
157
158     //armo los alcanzables directo
159     for (int j=-potenciaIntrinseca; j<=potenciaIntrinseca; j++){
160         //validacion de bordes y que no sea el mismo
161         if (validarBordes(dimensionMatriz, currentX, j)){
162             Nodo alcanzableDirecto = new Nodo(currentX + j, currentY, currentLevel);
163             alcanzables.add(alcanzableDirecto);
164         }
165
166         //validacion de bordes y que no sea el mismo
167         if (validarBordes(dimensionMatriz, currentY, j)){
168             Nodo alcanzableDirecto = new Nodo(currentX, currentY + j, currentLevel);
169             alcanzables.add(alcanzableDirecto);
170         }
171     }
172 }

```

```

164 //adiciono los alcanzables indirectos que surgen de utilizar
165 unidadesDePowerUpAdicionadas unidades de powerup
166 for(int unidadesDePowerUpAdicionadas=1;unidadesDePowerUpAdicionadas<=currentLevel;
167 unidadesDePowerUpAdicionadas++){
168     int potenciaTotal = potenciaIntrinseca + unidadesDePowerUpAdicionadas;
169
170     if(potenciaTotal>dimensionMatriz){
171         //fix para complejidad, de esta forma es O(dimension) todo el metodo.
172         //si potenciaTotal > dimensionMatriz, validarBordes va a dar false, asi que ni
173         //hace falta revisarlas
174         //veamos que como potenciaIntrinseca > 0 para cualquier nodo, entonces si
175         //unidadesDePowerUpAdicionadas > n-1
176         //entonces potenciaTotal > dimensionMatriz lo cual no tiene sentido, porque se
177         //podria saltar por fuera de los limites de la matriz
178         break;
179     }
180
181     if(validarBordes(dimensionMatriz, currentY, potenciaTotal)){
182         Nodo alcanzableIndirecto = new Nodo(currentX, currentY + potenciaTotal,
183         currentLevel - unidadesDePowerUpAdicionadas);
184         alcanzables.add(alcanzableIndirecto);
185     }
186
187     if(validarBordes(dimensionMatriz, currentY, -potenciaTotal)){
188         Nodo alcanzableIndirecto = new Nodo(currentX, currentY - potenciaTotal,
189         currentLevel - unidadesDePowerUpAdicionadas);
190         alcanzables.add(alcanzableIndirecto);
191     }
192
193     if(validarBordes(dimensionMatriz, currentX, potenciaTotal)){
194         Nodo alcanzableIndirecto = new Nodo(currentX + potenciaTotal, currentY,
195         currentLevel - unidadesDePowerUpAdicionadas);
196         alcanzables.add(alcanzableIndirecto);
197     }
198
199     if(validarBordes(dimensionMatriz, currentX, -potenciaTotal)){
200         Nodo alcanzableIndirecto = new Nodo(currentX - potenciaTotal, currentY,
201         currentLevel - unidadesDePowerUpAdicionadas);
202         alcanzables.add(alcanzableIndirecto);
203     }
204 }
205 return alcanzables;
206 }
207
208 private boolean validarBordes(int dimensionMatriz, int coordenada, int corrimiento) {
209     return (0<=coordenada+corrimiento) && (coordenada+corrimiento<dimensionMatriz) && (
210     corrimiento!=0);
211 }
212
213 /**
214 * Modificacion de bfs que busca un camino entre inicial y target(sin importar la
215 * coordenada level)
216 * es decir, cuando bfs encuentra un nodo w tal que las primeras 2 coordenadas de w
217 * coinciden con target, termina el algoritmo
218 * y reconstruye el camino en base a los predecesores, que se van guardando a medida que se
219 * realiza la expansion en anchura
220 * Teniendo en cuenta que bfs encuentra el camino mas corto en cantidad de aristas entre 2
221 * nodos, esto resuelve nuestro problema.
222 * @return Devuelve una lista de nodos indicando (posX, posY, gasto de PowerUp en el salto
223 * del predecesor de este nodo al actual)
224 */
225 public List<Nodo> caminoMinimo() {
226     LinkedList<Nodo> queue = new LinkedList<Nodo>();
227
228     queue.addLast(nodoInicial);
229     marcarVisitado(nodoInicial);
230
231     while(!queue.isEmpty()){
232         Nodo current = queue.removeFirst();
233         if(esDestino(current)){
234             //llegue a destino, termino la busqueda y devuelvo el resultado.
235             //System.out.println("Camino c <(origen) ,...(x, y, powerUtilizado)...,(destino)
236             >");
237             //armo el camino con los predecesores
238             return construirCaminoConPredecesores(current);
239         }
240     }
241 }

```



```

225         List<Nodo> alcanzables = obtenerAlcanzables(current);
226         for(Nodo alcanzable : alcanzables){
227             if(!estaVisitado(alcanzable)){
228                 marcarVisitado(alcanzable);
229                 asignarPredecesor(alcanzable, current);
230                 queue.addLast(alcanzable);
231             }
232         }
233     }
234     //como bfs recorre todos los nodos, llegamos a este punto unicamente si no existe nodo
        destino en el grafo
235     return null;
236 }
237
238 private LinkedList<Nodo> construirCaminoConPredecesores(Nodo current) {
239     //Tengamos en cuenta que en esta lista hay tuplas(x,y,z), x,y son las
240     //posiciones en el plano z, que indica, la cantidad de powerup disponible.
241     //Haciendo la resta entre el valor level de dos nodos v->w, se calcula el
242     //powerup utilizado para dicho salto entre v y w
243     //Definamos powerUpUsado(u,v) = level(v) - level(u)
244     //Asumamos que powerUpUsado(null, v) = 0;
245     //powerUpInicial - level(destino) te da la cantidad de powerUp usado total y
246     //level(destino) la cantidad de powerup disponible al salir del laberinto.
247     LinkedList<Nodo> camino = new LinkedList<Nodo>();
248     //armo una lista de nodos que indican el camino de origen a destino.
249     //como tengo el predecesor para cada nodo, puedo ir de atras hacia adelante
        encadenandolo.
250     Nodo nodo = current;
251     while(damePredecesor(nodo) != null){
252         Nodo predecesor = damePredecesor(nodo);
253         nodo.setLevel(predecesor.getLevel() - nodo.getLevel());
254         camino.addFirst(nodo);
255
256         //avanzo en el camino
257         nodo = predecesor;
258     }
259     nodo.setLevel(0);
260     camino.addFirst(nodo);
261     return camino;
262 }
263
264 private Nodo damePredecesor(Nodo nodo) {
265     return graph.get(nodo).getPredecesor();
266 }
267
268 private void asignarPredecesor(Nodo aMarcar, Nodo predecesor) {
269     graph.get(aMarcar).setPredecesor(predecesor);
270 }
271
272 private boolean estaVisitado(Nodo nodo) {
273     return graph.get(nodo).fueVisitado();
274 }
275
276 private void marcarVisitado(Nodo nodo){
277     graph.get(nodo).marcarVisitado();
278 }
279
280 private List<Nodo> obtenerAlcanzables(Nodo nodo){
281     return graph.get(nodo).getAlcanzables();
282 }
283
284 private boolean esDestino(Nodo current) {
285     //como no nos piden ninguna restriccion ni optimizacion sobre la utilizacion de
        powerup
286     //los niveles de estos nodos no son tenidos en cuenta en el algoritmo de busqueda para
        distinguir cuando llego a destino
287     return (current.getX() == nodoDestino.getX()) && (current.getY() == nodoDestino.getY())
        ;
288 }
289
290 }

```

6. Apéndice: Entregable e instrucciones de compilacion y testing

6.1. Estructura de directorios

- **ej1:** Contiene el codigo fuente en java del ejercicio 1, tanto como los scripts de compilacion nativa, testeo y graficacion, casos de tests, mediciones, y graficos.
- **ej2:** Contiene el codigo fuente en C++ del ejercicio 2 y su Makefile para compilar.
- **ej3:** Contiene el codigo fuente en java del ejercicio 3, tanto como los scripts de compilacion nativa, testeo y graficacion, casos de tests, mediciones y graficos.
- **informe:** Contiene los fuentes de latex, imagenes y codigo relevante junto al pdf del informe
- **casos:** Contiene el programa que genera los casos del ej2

6.2. Compilacion y ejecucion

- **ej1:** Ejecutando `./compilacionNativa.sh` se compila el programa, se crean y resuelven tests aleatorios tomando tiempos y se grafican en png los resultados.
- **ej2:** utilizando el Makefile y corriendo el ejecutable.
- **ej3:** Ejecutando `./compilacionNativa.sh` se compila el programa, se crean y resuelven tests aleatorios tomando tiempos y se grafican en png los resultados.

6.3. Generacion de tests aleatorios y toma de tiempos

- **ej1:** pasandole el parametro `-take-time <cant_repeticiones>` se repite la ejecucion `cant_repeticiones` veces tomando tiempo promedio en microsegundos . `-generate-tests <cards number> <randMin> <randMax>` genera casos aleatorios correspondientes y los arroja por stdout.
- **ej2:** Utilizando el programa especifico en la carpeta casos.
- **ej1:** pasandole el parametro `-take-time <cant_repeticiones>` se repite la ejecucion `cant_repeticiones` veces tomando tiempo promedio en microsegundos . `-generate-tests <dimension> <powerUp inicial>` genera casos aleatorios correspondientes y los arroja por stdout.y graficos.