DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Algoritmos y Estructura de Datos III

Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Iván Arcuschin	678/13	iarcuschin@gmail.com
Martín Jedwabny	885/13	martiniedva@gmail.com
José Massigoge	954/12	jmmassigoge@gmail.com
Lucas Puterman	830/13	lucasputerman@gmail.com

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) 4576-3359

http://www.fcen.uba.ar

Índice

1. Ejercicio 1 - Dakkar

1.1. Problema a resolver

El problema a resolver consiste en elegir que vehículo usar para cada etapa de un Rally Dakar (n etapas), teniendo en cuenta que las posibilidades son: una BMX, una motocross o un buggy arenero, y la cantidad de veces que podemos utilizar los últimos dos es limitada (K_m y K_b respectivamente.).

Para cada etapa, el tiempo que nos toma realizarla varía dependiendo de que vehículo elijamos. La elección de los vehículos para las diferentes etapas debe ser tal que minimice el tiempo total.

- Ejemplo 1:

```
2 etapas, 0 etapas máximo con la motocross, 1 etapa máximo con el buggy
etapa 1:
    - BMX: 6 minutos
    - motocross: 2 minutos
    - buggy: 3 minutos
etapa 2:
    - BMX: 2 minutos
    - motocross: 3 minutos
    - buggy: 8 minutos
```

- Solución del ejemplo 1:

```
Tiempo total: 5 minutos
etapa 1: buggy - 3 minutos
etapa 2: BMX - 2 minutos
```

- Ejemplo 2:

```
5 etapas, 2 etapas máximo con la motocross, 1 etapa máximo con el buggy
etapa 1:
    - BMX: 10 minutos
    - motocross: 4 minutos
    - buggy: 3 minutos
etapa 2:
    - BMX: 15 minutos
    - motocross: 3 minutos
    - buggy: 6 minutos
etapa 3:
    - BMX: 6 minutos
    - motocross: 5 minutos
    - buggy: 5 minutos
etapa 4:
    - BMX: 7 minutos
    - motocross: 6 minutos
    - buggy: 2 minutos
etapa 5:
    - BMX: 12 minutos
    - motocross: 2 minutos
    - buggy: 6 minutos
```

- Solución del ejemplo 2:

```
Tiempo total: 21 minutos
etapa 1: buggy - 3 minutos
etapa 2: motocross - 3 minutos
etapa 3: BMX - 6 minutos
etapa 4: BMX - 7 minutos
etapa 5: motocross - 2 minutos
```

1.2. Resolución planteada

Al ser un problema de optimización, vamos a resolverlo usando la técnica algorítmica de `Programación Dinámica`.

Pero antes de desarrollar el algoritmo, veamos que el problema cumpla con los requisitos necesarios para que se pueda aplicar la técnica mencionada[1]:

1.2.1. Subestructura Óptima

Como mostramos brevemente antes, una solución para un problema de n etapas usando a lo sumo i motos y j buggies consiste en:

- Un vector de vehículos $V_{n,i,j}$ de tamaño n, donde $V_{n,i,j}[h]$ indica el vehículo elegido para la etapa h en la solución.
- El tiempo total obtenido con los vehículos elegidos: $T_{n,i,j} = \sum_{h=1}^{n} costo(V_{n,i,j}[h], h)$. Donde $costo(v, h)$ es el costo de usar el vehículo v en la etapa h.

Y una solución $V_{n,i,j}$ es optima si:

$$(\forall\ W_{n,i,j})\ \sum_{h=1}^{n} costo(V_{n,i,j}[h], h) \leq \sum_{h=1}^{n} costo(W_{n,i,j}[h], h)$$

Sean $V_{n,i,j}$, $V_{n-1,i-1,j}$, $V_{n-1,i,j-1}$ y $V_{n-1,i,j}$ soluciones optimas. Entonces:

[1]Los requisitos necesarios son extraidos del libro: Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.The MIT Press, 2005

- $V_{n,i,j}$ es la solución óptima para n etapas usando a lo sumo i motos y j buggies.
- $V_{n-1,i,j}$ es la solución óptima para $n-1$ etapas usando a lo sumo i motos y j buggies.
- $V_{n-1,i-1,j}$ es la solución óptima para $n-1$ etapas usando a lo sumo $i-1$ motos y j buggies.
- $V_{n-1,i,j-1}$ es la solución óptima para $n-1$ etapas usando a lo sumo i motos y $j-1$ buggies.

Ahora, vamos a mostrar que la solución óptima $V_{n,i,j}$ contiene alguna de las soluciones óptimas $V_{n-1,i,j}$, $V_{n-1,i-1,j}$ o $V_{n-1,i,j-1}$.

Lema 1.1. *Dada una solución óptima $V_{i,j}$ (cuyo tiempo es $T_{n,i,j}$), o bien:*

1. $T_{n,i,j} = T_{n-1,i,j} + costo(BMX, n) \implies T_{n-1,i,j}$ *es el tiempo de la solución óptima $V_{n-1,i,j}$ para $n-1$ etapas usando a lo sumo i motos y j buggies, o bien*
2. $T_{n,i,j} = T_{n-1,i-1,j} + costo(MOTO, n) \implies T_{n-1,i-1,j}$ *es el tiempo de la solución óptima $V_{n-1,i-1,j}$ para $n-1$ etapas usando a lo sumo $i-1$ motos y j buggies, o bien*
3. $T_{n,i,j} = T_{n-1,i,j-1} + costo(BUGGY, n) \implies T_{n-1,i,j-1}$*es el tiempo de la solución óptima $V_{n-1,i,j-1}$ para $n-1$ etapas usando a lo sumo i motos y $j-1$ buggies.*

Demostración. Planteamos los 3 casos posibles:

1. Si $V_{n,i,j}[n] = BMX \to V_{n-1,i,j} = \{V_{n,i,j}[1],\ \ldots,\ V_{n,i,j}[n-1]\}$ es óptima.
2. Si $V_{n,i,j}[n] = MOTO \to V_{n-1,i-1,j} = \{V_{n,i,j}[1],\ \ldots,\ V_{n,i,j}[n-1]\}$ es óptima.
3. Si $V_{n,i,j}[n] = BUGGY \to V_{n-1,i,j-1} = \{V_{n,i,j}[1],\ \ldots,\ V_{n,i,j}[n-1]\}$ es óptima.

1. Antes que nada, sabemos que $V_{n-1,i,j}$ es válida porque al haber una BMX en la última etapa de $V_{n,i,j}$ sabemos que al considerar las $n-1$ etapas tenemos a lo sumo i motos y a lo sumo j buggies.

   Ahora, por absurdo, supongamos que $V_{n-1,i,j}$ no es óptima, o sea que $\exists\, W_{n-1,i,j} = \{w_1,\ \ldots,\ w_{n-1}\}$ tal que $T_W < T_{n-1,i,j}$. Si consideramos ahora $Z_{n,i,j} = \{w_1,\ \ldots,\ w_{n-1},\ BMX\}$, vemos que es una solución valida para n etapas, ya que W tenía a lo sumo i motos y a lo sumo j buggies en sus $n-1$ etapas, y Z agrega una BMX en la etapa n. Pero entonces,

   $$\begin{aligned} T_Z &= \sum_{h=1}^{n-1} costo(W_{n-1,i,j}[h], h) + costo(BMX, n) = T_W + costo(BMX, n) \\ &< T_{n-1,i,j} + costo(BMX, n) = \sum_{h=1}^{n-1} costo(V_{n,i,j}[h], h) + costo(BMX, n) \\ &= \sum_{h=1}^{n} costo(V_{n,i,j}[h], h) = T_{n,i,j} \end{aligned}$$

   Absurdo, $V_{n,i,j}$ era óptima y Z es una solución para n etapas con a lo sumo i motos y a lo sumo j buggies que tiene un tiempo menor estricto.

   Luego, el absurdo viene de suponer que $V_{n-1,i,j}$ no es óptima.

2. Analogamente al caso anterior, sabemos que $V_{n-1,i-1,j}$ es válida porque al haber una $MOTO$ en la última etapa de $V_{n,i,j}$ sabemos que al considerar las $n-1$ etapas tenemos a lo sumo $i-1$ motos y a lo sumo j buggies.

   Ahora, por absurdo, supongamos que $V_{n-1,i-1,j}$ no es óptima, o sea que $\exists\, W_{n-1,i-1,j} = \{w_1,\ \ldots,\ w_{n-1}\}$ tal que $T_W < T_{n-1,i-1,j}$. Si consideramos ahora $Z_{n,i,j} = \{w_1,\ \ldots,\ w_{n-1},\ MOTO\}$, vemos que es una solución valida para n etapas, ya que W tenía a lo sumo $i-1$ motos y a lo sumo j buggies en

sus $n-1$ etapas, y Z agrega una $MOTO$ en la etapa n, por lo que ahora tenemos a lo sumo i motos. Pero entonces,

$$T_Z = \sum_{h=1}^{n-1} costo(W_{n-1,i-1,j}[h], h) + costo(MOTO, n) = T_W + costo(MOTO, n)$$
$$< T_{n-1,i-1,j} + costo(MOTO, n) = \sum_{h=1}^{n-1} costo(V_{n,i,j}[h], h) + costo(MOTO, n)$$
$$= \sum_{h=1}^{n} costo(V_{n,i,j}[h], h) = T_{n,i,j}$$

Absurdo, $V_{n,i,j}$ era óptima y Z es una solución para n etapas con a lo sumo i motos y a lo sumo j buggies que tiene un tiempo menor estricto.

Luego, el absurdo viene de suponer que $V_{n-1,i-1,j}$ no es óptima.

3. Idem caso 2.

□

1.2.2. Subproblemas superpuestos

Queremos mostrar que la solución óptima $V_{n,i,j}$ se obtiene a partir de las soluciones óptimas $V_{n-1,i,j}$, $V_{n-1,i-1,j}$ o $V_{n-1,i,j-1}$.

Lema 1.2. *Dadas las soluciones óptimas $V_{n-1,i,j}$, $V_{n-1,i-1,j}$ y $V_{n-1,i,j-1}$ (cuyos tiempos son $T_{n-1,i,j}$, $T_{n-1,i-1,j}$, $T_{n-1,i,j-1}$ respectivamente):*

$$T_{n,i,j} = min(T_{n-1,i,j} + costo(BMX, n), T_{n-1,i-1,j} + costo(MOTO, n), T_{n-1,i,j-1} + costo(BUGGY, n))$$

Donde $T_{n,i,j}$ es el tiempo de la solución óptima $V_{n,i,j}$.

Demostración. Supongamos, por absurdo, que $V_{n,i,j}$ no es óptima. O sea que $\exists\ W_{n,i,j} = \{w_1, \ldots, w_n\}$ tal que $T_W < T_{n,i,j}$.

Ahora, hay solo 3 posibilidades para el vehículo en $W_{n,i,j}[n]$ (la n-esima etapa de $W_{n,i,j}$):

1. $W_{n,i,j}[n] = BMX$
2. $W_{n,i,j}[n] = MOTO$
3. $W_{n,i,j}[n] = BUGGY$

Veamos que pasa en cada uno de ellos:

1. Sea $Z_{n-1,i,j} = \{w_1, \ldots, w_{n-1}\}$ y sea T_Z el tiempo de $Z_{n-1,i,j}$. Sabemos que $Z_{n-1,i,j}$ es una solución valida ya que como $W_{n,i,j}$ tiene a lo sumo i motos y j buggies en la n-esima etapa, y esa etapa tiene una BMX, entonces $W_{n,i,j}$ tiene a lo sumo i motos y j buggies en sus $n-1$ etapas restantes. Luego:

$$T_W = \sum_{h=1}^{n} costo(W_{n,i,j}[h], h) = \sum_{h=1}^{n-1} costo(W_{n,i,j}[h], h) + costo(BMX, n)$$
$$= T_Z + costo(BMX, n) < T_{n,i,j}$$

Pero como T_W es menor que $T_{n,i,j}$, en particular es menor que $T_{n-1,i,j} + costo(BMX, n)$. Entonces:

$$T_W = T_Z + costo(BMX, n) < T_{n-1,i,j} + costo(BMX, n)$$
$$\iff T_Z < T_{n-1,i,j}$$

Absurdo, ya que partimos de que $T_{n-1,i,j}$ era el tiempo de la solución óptima $V_{n-1,i,j}$. Luego, el absurdo viene de suponer que $V_{n,i,j}$ no es óptima.

2. Sea $Z_{n-1,i-1,j} = \{w_1, \ldots, w_{n-1}\}$ y sea T_Z el tiempo de $Z_{n-1,i-1,j}$. Sabemos que $Z_{n-1,i-1,j}$ es una solución valida ya que como $W_{n,i,j}$ tiene a lo sumo i motos y j buggies en la n-esima etapa, y esa etapa tiene una $MOTO$, entonces $W_{n,i,j}$ tiene a lo sumo $i-1$ motos y j buggies en sus $n-1$ etapas restantes. Luego:

$$\begin{aligned} T_W &= \sum_{h=1}^{n} costo(W_{n,i,j}[h], h) = \sum_{h=1}^{n-1} costo(W_{n,i,j}[h], h) + costo(MOTO, n) \\ &= T_Z + costo(MOTO, n) < T_{n,i,j} \end{aligned}$$

Pero como T_W es menor que $T_{n,i,j}$, en particular es menor que $T_{n-1,i-1,j} + costo(MOTO, n)$. Entonces:

$$\begin{aligned} T_W = T_Z + costo(MOTO, n) &< T_{n-1,i-1,j} + costo(MOTO, n) \\ \iff T_Z &< T_{n-1,i-1,j} \end{aligned}$$

Absurdo, ya que partimos de que $T_{n-1,i-1,j}$ era el tiempo de la solución óptima $V_{n-1,i-1,j}$. Luego, el absurdo viene de suponer que $V_{n,i,j}$ no es óptima.

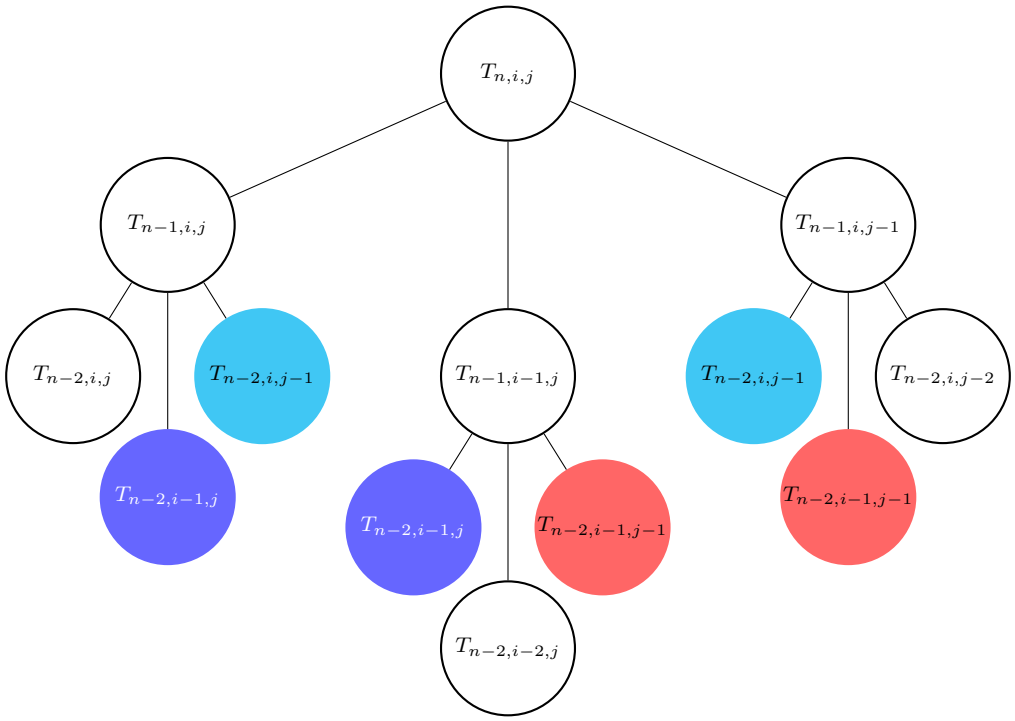
3. Analoga al caso 2.

□

Como vimos en el **Lema 1.2**, para resolver un problema con n etapas y con a lo sumo i motos y j buggies, necesitamos la solución óptima de los siguientes subproblemas:

1. $n-1$ etapas, con a lo asumo i motos y j buggies.
2. $n-1$ etapas, con a lo asumo $i-1$ motos y j buggies.
3. $n-1$ etapas, con a lo asumo i motos y $j-1$ buggies.

De la misma manera para resolver los subproblemas enumerados previamente, necesitamos la solución óptima de otros subproblemas. A continuación se detalla la dependencia de cada subproblema:



Como surge de la imagen, hay subproblemas cuya solución es requerida por mas de un subproblema, generando una superposición de los mismos. De esta manera, queda evidenciado la existencia de *Subproblemas superpuestos* en este problema.

1.2.3. Formulación recursiva

A continuación escribimos la solución del problema en términos de una función recursiva, donde $Km = i$ y $Kb = j$:

$$T_{n,i,j} = \begin{cases} costo(BMX, 1) & : n = 1 \wedge i = 0 \wedge j = 0 \\ min(costo(BMX, 1), costo(MOTO, 1)) & : n = 1 \wedge i > 0 \wedge j = 0 \\ min(costo(BMX, 1), costo(BUGGY, 1)) & : n = 1 \wedge i = 0 \wedge j > 0 \\ min(costo(BMX, 1), costo(MOTO, 1), costo(BUGGY, 1)) & : n = 1 \wedge i > 0 \wedge j > 0 \\ T_{n-1,0,0} + costo(BMX, n) & : n > 1 \wedge i = 0 \wedge j = 0 \\ min(T_{n-1,i,0} + costo(BMX, n), T_{n-1,i-1,0} + costo(MOTO, n)) & : n > 1 \wedge i > 0 \wedge j = 0 \\ min(T_{n-1,0,j} + costo(BMX, n), T_{n-1,0,j-1} + costo(BUGGY, n)) & : n > 1 \wedge i = 0 \wedge j > 0 \\ min(T_{n-1,i,j} + costo(BMX, n), T_{n-1,i-1,j} + costo(MOTO, n), & : n > 1 \wedge i > 0 \wedge j > 0 \\ \quad T_{n-1,i,j-1} + costo(BUGGY, n)) & \end{cases}$$

Habiendo justificado la utilización de la técnica algorítmica de `Programación Dinámica`, el algoritmo utilizado para resolver el problema, que sigue una estrategia bottom-up, se desarrolla de la siguiente forma:

- Creamos un vector *costos* de tamaño igual a n, donde en el indice i se almacenan los tiempos de la BMX, la moto y el buggy para la etapa i.

  Observación: Los indices de las etapas a partir de ahora los vamos a representar de 0 a $n-1$, en vez de 1 a n como veniamos haciendo. De esta forma esperamos que quede más claro la descripción del algoritmo, ya que coincide con la forma de acceder a las estructuras utilizadas.

- Creamos un vector de matrices de tamaño $n * (Km + 1) * (Kb + 1)$ celdas, donde cada celda tiene atributos de `tiempo`, `vehículo` y `antecesor`, que nos permitiran más adelante reconstruir la solucion final.

- Completamos la primera posición del vector de matrices (la primera etapa), de la siguiente manera:

  1. En la celda $(0, 0, 0)$ ponemos el costo del BMX para la etapa 0.
  2. Luego para el caso de la celda $(0, i, 0)$ comparamos los costos para la etapa 0 de la BMX y de la MOTO, asignando al atributo `tiempo` el menor de ambos, y al atributo `vehículo` el vehículo correspondiente a ese costo, siendo el atributo `antecesor` un valor invalido debido a que estamos en la primera etapa. El caso $(0, 0, j)$ es análogo.
  3. Para los casos donde $i, j > 0$, la comparación se debe hacer con los costos de todos los vehículos de la etapa 0, asignando a `tiempo` el tiempo menor de todos, y a `vehículo` el vehículo correspondiente a ese costo.

- Luego, si tomamos una etapa h cualquiera (con $0 < h < n$) podemos completar de la siguiente forma:

  1. En la celda $(h, 0, 0)$ ponemos el costo de la BMX para la etapa h mas el tiempo en la celda $(h-1, 0, 0)$.
  2. Luego para el caso de la celda $(h, i, 0)$ comparamos los costos para la etapa h de la BMX mas el tiempo en la celda $(h-1, i, 0)$ y de la MOTO en la etapa h mas el tiempo en la celda $(h-1, i-1, 0)$, asignando a `tiempo` el menor de ambos, y a `vehículo` el vehículo correspondiente a ese costo, siendo `antecesor` $(i, 0)$ si el vehículo seleccionado es un BMX y $(i-1, 0)$ si el vehículo seleccionado es una MOTO. El caso $(h, 0, j)$ es análogo.
  3. Para los casos donde $i, j > 0$, comparamos los costos para la etapa h de la BMX mas el tiempo en la celda $(h-1, i, j)$, de la MOTO en la etapa h mas el tiempo en la celda $(h-1, i-1, j)$, y del BUGGY en la etapa h mas el tiempo en la celda $(h-1, i, j-1)$ asignando a `tiempo` el costo menor de todos, y a `vehículo` el vehículo correspondiente a ese costo, y a `antecesor` (i, j) si el vehículo seleccionado es un BMX, $(i-1, j)$ si el vehículo seleccionado es una MOTO, y $(i, j-1)$ si el vehículo seleccionado es un BUGGY.

- Al finalizar, el tiempo optimo lo obtenemos a partir de la posición $(n-1,i,j)$ y los vehículos correspondiente a cada etapa, los obtenemos a partir del antecesor de $(n-1,i,j)$, donde el $vehiculo[n-1]$ es igual al vehículo en $(n-1,i,j)$, $vehiculo[n-2]$ es igual al vehículo en el antecesor de $(n-1,i,j)$, y así sucesivamente hasta obtener los vehículos para cada etapa.

1.2.4. Pseudocódigo

En pseudocódigo nos queda:

```
Creamos una vector costos de tamaño igual a n, donde en el indice i se almacenan los
    tiempos de la bmx, la moto y el buggy para la etapa i-1.
Si Kb > n:
    Pongo Kb = n
Si Km > n:
    Pongo Km = n
Creamos un vector de matrices, infoEtapas, de n*(Km+1)*(Kb+1) celdas para almacenar
    las soluciones parciales.
Completamos la primera matriz del vector de la siguiente manera:
Para i = 0 hasta Km
    Para j = 0 hasta Kb
            Si i = 0 y j = 0 entonces:
                infoEtapas[0,0,0].tiempo = costo(BMX,0)
                infoEtapas[0,0,0].vehiculo = BMX
            Sino Si i = 0 entonces:
                 Si costo(BMX,0) < costo(BUGGY,0) entonces:
                    infoEtapas[0,0,j].tiempo = costo(BMX,0)
                    infoEtapas[0,0,j].vehiculo = BMX
                 Sino:
                    infoEtapas[0,0,j].tiempo = costo(BUGGY,0)
                    infoEtapas[0,0,j].vehiculo = BUGGY
             Sino Si j = 0 entonces:
                 Si costo(BMX,0) < costo(MOTO,0) entonces:
                    infoEtapas[0,i,0].tiempo = costo(BMX,0)
                    infoEtapas[0,i,0].vehiculo = BMX
                 Sino:
                    infoEtapas[0,i,0].tiempo = costo(MOTO,0)
                    infoEtapas[0,i,0].vehiculo = MOTO
             Sino:
                tBMX = costo(BMX,0)
                tMOTO = costo(MOTO,0)
                tBUGGY = costo(BUGGY,0)
                Si tBMX < tMOTO y tBMX < tBUGGY  entonces:
                    infoEtapas[0,i,j].tiempo = tBMX
                    infoEtapas[0,i,j].vehiculo = BMX
                Sino Si tMOTO < tBUGGY:
                    infoEtapas[0,i,j].tiempo = tMOTO
                    infoEtapas[0,i,j].vehiculo = MOTO
                Sino:
                   infoEtapas[0,i,j].tiempo = tBUGGY
                   infoEtapas[0,i,j].vehiculo = BUGGY
    Fin para todo
Fin para todo

                                                                    (Continua)
```

```
Completamos el resto de las matrices del vector:
Para h = 1 hasta n-1
        Para i = 0 hasta Km
            Para j = 0 hasta Kb
                Si i = 0 y j = 0 entonces:
                    infoEtapas[h,0,0].tiempo = infoEtapas[h-1,0,0].tiempo + costo(BMX,h)
                    infoEtapas[h,0,0].vehiculo = BMX
                    infoEtapas[h,0,0].antecesor = (0,0)
                Sino Si i = 0 entonces:
                    thBMX = infoEtapas[h-1,0,j].tiempo + costo(BMX,h)
                    thBUGGY = infoEtapas[h-1,0,j-1].tiempo + costo(BUGGY,h)
                    Si  thBMX < thBUGGY  entonces:
                        infoEtapas[h,0,j].tiempo = thBMX
                        infoEtapas[h,0,j].vehiculo = BMX
                        infoEtapas[h,0,j].antecesor = (0,j)
                    Sino:
                        infoEtapas[h,0,j].tiempo = thBUGGY
                        infoEtapas[h,0,j].vehiculo = BUGGY
                        infoEtapas[h,0,j].antecesor = (0,j-1)
                Sino Si j = 0 entonces:
                    thBMX = infoEtapas[h-1,i,0].tiempo + costo(BMX,h)
                    thMOTO = infoEtapas[h-1,i-1,0].tiempo + costo(MOTO,h)
                     Si costo(BMX,0) < costo(MOTO,0) entonces:
                        infoEtapas[h,i,0].tiempo = thBMX
                        infoEtapas[h,i,0].vehiculo = BMX
                        infoEtapas[h,i,0].antecesor = (i,0)
                    Sino:
                        infoEtapas[h,i,0].tiempo = thMOTO
                        infoEtapas[h,i,0].vehiculo = MOTO
                        infoEtapas[h,i,0].antecesor = (i-1,0)
                Sino:
                    thBMX = infoEtapas[h-1,i,j].tiempo + costo(BMX,h)
                    thMOTO = infoEtapas[h-1,i-1,j].tiempo + costo(MOTO,h)
                    thBUGGY = infoEtapas[h-1,i,j-1].tiempo + costo(BUGGY,h)
                    Si thBMX < thMOTO y thBMX < thBUGGY  entonces:
                        infoEtapas[h,i,j].tiempo = thBMX
                        infoEtapas[h,i,j].vehiculo = BMX
                        infoEtapas[h,i,j].antecesor = (i,j)
                   Sino Si thMOTO < thBUGGY:
                        infoEtapas[h,i,j].tiempo = thMOTO
                        infoEtapas[h,i,j].vehiculo = MOTO
                        infoEtapas[h,i,j].antecesor = (i-1,j)
                  Sino:
                        infoEtapas[h,i,j].tiempo = thBUGGY
                        infoEtapas[h,i,j].vehiculo = BUGGY
                        infoEtapas[h,i,j].antecesor = (i,j-1)
           Fin para todo
      Fin para todo
 Fin para todo

Recontruimos la solución a partir de los antecesores
    de la posición infoEtapas[n-1,Km,Kb]
```

1.3. Complejidad propuesta

Como se desprende del pseudocódigo del punto anterior, el algoritmo que proponemos tiene 6 ciclos (uno simple, uno de 2 ciclos anidados y otro de 3 ciclos anidados). Vamos a verlos uno por uno:

1. Completar el vector *costos* con los costos de cada vehículo para cada etapa implica repetir la operación de almacenar datos n veces, siendo la complejidad de la operación de almacenar constante por ser un vector para el cual reservamos n posiciones. Por lo cual, la complejidad de este ciclo es de $\Theta(n)$.

2. Completar la primera matriz del vector *infoEtapas* implica iterar $(Km + 1) * (Kb + 1)$ veces. Las operaciones dentro del ciclo son accesos, almacenamientos a un vector, y comparaciones, por lo cual todas las operaciones son constantes. Por lo tanto, la complejidad de este ciclo es de $\Theta((Km + 1) * (Kb + 1)) \in \Theta(Km * Kb)$.

3. Completamos el resto de las matrices del vector *infoEtapas* lo cual implica iterar $(n - 1) * (Km + 1) * (Kb + 1)$ veces. Las operaciones dentro del ciclo son accesos, almacenamientos a un vector, y comparaciones, por lo cual todas las operaciones son constantes. Por lo tanto, la complejidad de este ciclo es de $\Theta((n - 1) * (Km + 1) * (Kb + 1)) \in \Theta(n * Km * Kb)$.

4. El ultimo ciclo consiste en accesos a la infoEtapas para reconstruir el vector vehículos solución, desde la ultima posición. Para construir el vector vehículos solución iteramos n veces, haciendo en cada iteración operaciones de costo constante. La complejidad de este ciclo es de $\Theta(n)$.

Nos queda por ver las operaciones que están por fuera de los ciclos, las cuales son dos creaciones de vectores (uno para los *costos* y otro para rearmar los *vehiculos* de la solución final) y una creación de un vector de matrices. La creación de los vectores tiene una complejidad de $\Theta(n)$, mientras que la creación del vector de matrices tiene una complejidad de $\Theta((n - 1) * (Km + 1) * (Kb + 1)) \in \Theta(n * Km * Kb)$.

A partir de lo que expusimos previamente, el calculo de complejidad es: $\Theta(n + Km * Kb + n * Km * Kb + n + n + n * Km * Kb + n)$, es decir $\Theta(n * Km * Kb) \in \mathcal{O}(n * Km * Kb)$ lo cual cumple con la complejidad temporal solicitada.

1.4. Implementación en C++

```
#include <vector>
#include <iostream>
using namespace std;

// Structs y Typedefs
// para indexar los arrays de costos y otras operaciones con vehiculos
#define BMX         0
#define MOTO        1
#define BUGGY       2

// struct que contiene el costo de una etapa con diferentes vehiculos
struct costoEtapa {
    costoEtapa() : bmx(0), moto(0), buggy(0) {}

    costoEtapa(int t1, int t2, int t3) : bmx(t1), moto(t2), buggy(t3) {}

    int costo(int vehiculo) {
        if (vehiculo == BMX) {
            return bmx;
        } else if (vehiculo == MOTO) {
            return moto;
        } else {
            return buggy;
        }
    }

    int bmx;
    int moto;
    int buggy;
};

// celda de la Tabla
struct celda {
    celda() : tiempo(0), antecesor(-1,-1), vehiculo(-1) {}

    pair<int, int> antecesor;
    int vehiculo;
    int tiempo;
};

typedef vector<celda> Vec;
typedef vector<Vec> Tabla;
typedef vector<Tabla> HiperTabla;

// Prototipado de funciones
void resolver(int n, int km, int kb, vector<costoEtapa>& costos);
int min(int a, int b);
int min(int a, int b, int c);

// Implementacion.
int main() {
    // cantidad de etapas
```

```
    int n;
    cin >> n;
    // cantidad de veces que puedo usar la moto
    int km;
    cin >> km;
    // cantidad de veces que puedo usar el buggy
    int kb;
    cin >> kb;

    // limito la cantidad de motos a la cantidad de etapas
    if(km > n) {
        km = n;
    }
    // limito la cantidad de buggys a la cantidad de etapas
    if(kb > n) {
        kb = n;
    }

    // cargo los costos
    vector<costoEtapa> costos(n, costoEtapa());
    int bmx, moto, buggy;
    for(int i = 0; i < n; i++) {
        cin >> bmx;
        cin >> moto;
        cin >> buggy;
        costoEtapa costo = costoEtapa(bmx, moto, buggy);
        costos[i] = costo;
    }

    resolver(n, km, kb, costos);
}

void resolver(int n, int km, int kb, vector<costoEtapa>& costos) {
    // creo la tabla de celdaes parciales
    // las etapas van de 0 a n-1
    HiperTabla matriz(n, Tabla(km+1, Vec(kb+1, celda())));

    // inicializo n = 0
    for (int i = 0; i <= km; i++) {
        for (int j = 0; j <= kb; j++) {
            if (i == 0 && j == 0) {
                matriz[0][i][j].tiempo = costos[0].costo(BMX);
                matriz[0][i][j].vehiculo = BMX;
            } else if (i == 0) { // j != 0
                int aux1 = costos[0].costo(BMX);
                int aux2 = costos[0].costo(BUGGY);
                matriz[0][i][j].tiempo = min(aux1, aux2);
                // guardo el vehiculo
                if (matriz[0][i][j].tiempo == aux1) {
                    matriz[0][i][j].vehiculo = BMX;
                } else {
                    matriz[0][i][j].vehiculo = BUGGY;
                }
            } else if (j == 0) { // i == 0
```

```
                int aux1 = costos[0].costo(BMX);
                int aux2 = costos[0].costo(MOTO);
                matriz[0][i][j].tiempo = min(aux1, aux2);
                // guardo el vehiculo
                if (matriz[0][i][j].tiempo == aux1) {
                    matriz[0][i][j].vehiculo = BMX;
                } else {
                    matriz[0][i][j].vehiculo = MOTO;
                }
            } else {
                int aux1 = costos[0].costo(BMX);
                int aux2 = costos[0].costo(MOTO);
                int aux3 = costos[0].costo(BUGGY);
                matriz[0][i][j].tiempo = min(aux1, aux2, aux3);
                // guardo el vehiculo
                if (matriz[0][i][j].tiempo == aux1) {
                    matriz[0][i][j].vehiculo = BMX;
                } else if (matriz[0][i][j].tiempo == aux2) {
                    matriz[0][i][j].vehiculo = MOTO;
                } else {
                    matriz[0][i][j].vehiculo = BUGGY;
                }
            }
        }
    }

    for (int h = 1; h < n; h++) {
        for (int i = 0; i <= km; i++) {
            for (int j = 0; j <= kb; j++) {
                if (i == 0 && j == 0) {
                    matriz[h][i][j].tiempo = costos[h].costo(BMX) +
                                             matriz[h-1][i][j].tiempo;
                    matriz[h][i][j].antecesor = make_pair(i,j);
                    matriz[h][i][j].vehiculo = BMX;
                } else if (i == 0) { // j != 0
                    int aux1 = costos[h].costo(BMX) +
                                             matriz[h-1][i][j].tiempo;
                    int aux2 = costos[h].costo(BUGGY) +
                                             matriz[h-1][i][j-1].tiempo;
                    matriz[h][i][j].tiempo = min(aux1, aux2);
                    // guardo el antecesor y el vehiculo
                    if (matriz[h][i][j].tiempo == aux1) {
                        matriz[h][i][j].antecesor = make_pair(i,j);
                        matriz[h][i][j].vehiculo = BMX;
                    } else {
                        matriz[h][i][j].antecesor = make_pair(i,j-1);
                        matriz[h][i][j].vehiculo = BUGGY;
                    }
                } else if (j == 0) { // i == 0
                    int aux1 = costos[h].costo(BMX) +
                                             matriz[h-1][i][j].tiempo;
                    int aux2 = costos[h].costo(MOTO) +
                                             matriz[h-1][i-1][j].tiempo;
                    matriz[h][i][j].tiempo = min(aux1, aux2);
```

```
                    // guardo el antecesor y el vehiculo
                    if (matriz[h][i][j].tiempo == aux1) {
                        matriz[h][i][j].antecesor = make_pair(i,j);
                        matriz[h][i][j].vehiculo = BMX;
                    } else {
                        matriz[h][i][j].antecesor = make_pair(i-1,j);
                        matriz[h][i][j].vehiculo = MOTO;
                    }
                } else {
                    int aux1 = costos[h].costo(BMX) +
                                        matriz[h-1][i][j].tiempo;
                    int aux2 = costos[h].costo(MOTO) +
                                        matriz[h-1][i-1][j].tiempo;
                    int aux3 = costos[h].costo(BUGGY) +
                                        matriz[h-1][i][j-1].tiempo;
                    matriz[h][i][j].tiempo = min(aux1, aux2, aux3);
                    // guardo el antecesor y el vehiculo
                    if (matriz[h][i][j].tiempo == aux1) {
                        matriz[h][i][j].antecesor = make_pair(i,j);
                        matriz[h][i][j].vehiculo = BMX;
                    } else if (matriz[h][i][j].tiempo == aux2) {
                        matriz[h][i][j].antecesor = make_pair(i-1,j);
                        matriz[h][i][j].vehiculo = MOTO;
                    } else {
                        matriz[h][i][j].antecesor = make_pair(i,j-1);
                        matriz[h][i][j].vehiculo = BUGGY;
                    }
                }
            }
        }
    }

    // reconstruyo la solucion
    vector<int> vehiculos(n, -1);
    celda aux = matriz[n-1][km][kb];
    for (int h = n-1; h > 0; h--) {
        vehiculos[h] = aux.vehiculo;
        aux = matriz[h-1][aux.antecesor.first][aux.antecesor.second];
    }
    vehiculos[0] = aux.vehiculo;

    // muestro la solucion por stdout
    cout << matriz[n-1][km][kb].tiempo << "␣";
    for(int i = 0; i < n; i++) {
        cout << vehiculos[i]+1 << "␣";
    }
    cout << endl;
}

int min(int a, int b) {
    return a < b ? a : b;
}

int min(int a, int b, int c) {
```

```
    return min(min(a,b),c);
}
```

1.5. Experimentación computacional

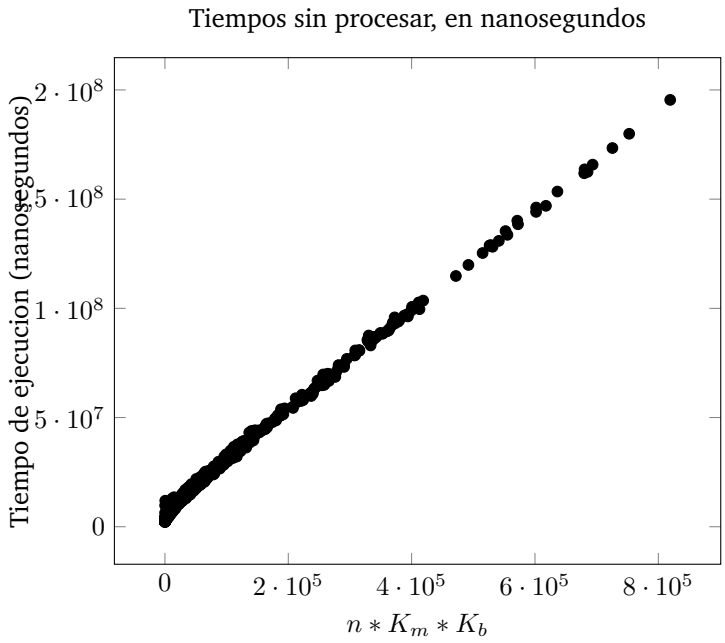
La función que utilizamos para llevar a cabo las mediciones fue `std::clock`[2]. La unidad temporal que utilizamos para este ejercicio fue nanosegundos. La complejidad teórica calculada es de $\mathcal{O}(n * K_b * K_m)$

1.5.1. Experimentación con instancias aleatorias

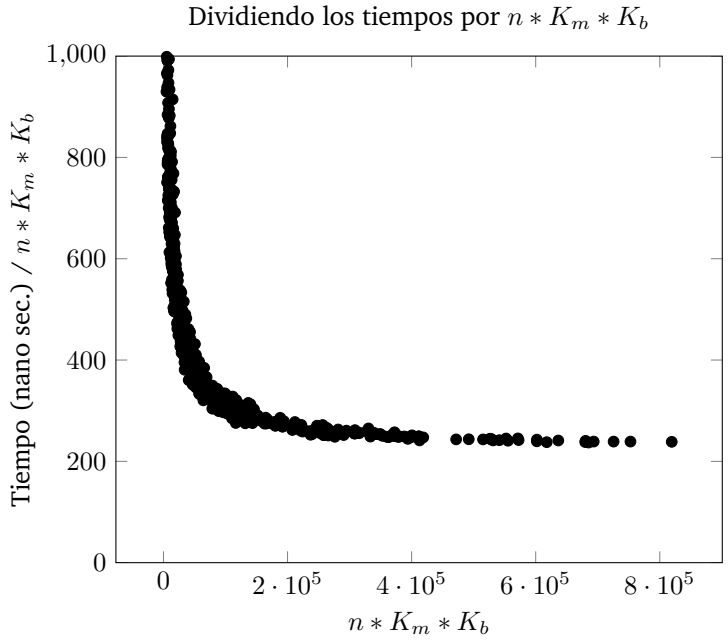
Para generar las instancias aleatorias utilizamos la función `std::rand`[3] con determinados intervalos de valores para la variables, para obtener instancias coherentes. El detalle de intervalos es el siguiente:

- Cantidad de etapas (n): $1 \leq n \leq 100$
- Costo maximo para bmx (C_{bmx}): $1 \leq C_{bmx} \leq 1.000$
- Cantidad de veces que se usa la Moto (K_m): $0 \leq K_m \leq n$
- Cantidad de veces que se usa el Buggy (K_b): $0 \leq K_b \leq n$

Generamos 100 instancias aleatorias para cada n, variando aleatoriamente el K_m y el K_b cada 10 muestras, luego promediadas. Su medición temporal, arroja el siguiente resultado:



[2]Referencia `http://en.cppreference.com/w/cpp/chrono/c/clock`
[3]Referencia `http://en.cppreference.com/w/cpp/numeric/random/rand`



A continuación, adjuntamos una tabla con los últimos 20 valores obtenidos en las instancias aleatorias, teniendo en cuenta que los casos fueron previamente ordenados según el tamaño (n):

Etapas(n)	K_m	K_b	Tiempo(nanosegundos)	Tiempo(nanosegundos) / ($n * K_m * K_b$)
99	72	37	70072900.00	265.69
99	12	73	29759500.00	343.15
99	31	7	11082100.00	515.85
99	49	81	97282600.00	247.58
99	69	45	78407200.00	255.06
99	79	87	163650100.00	240.51
99	69	21	40036000.00	279.09
99	98	59	138501600.00	241.95
99	83	17	43269100.00	309.75
99	60	89	128942200.00	243.90
100	26	40	32398500.00	311.52
100	12	17	11816700.00	579.25
100	69	37	67029800.00	262.55
100	47	19	28339800.00	317.35
100	51	19	29314000.00	302.51
100	14	49	22309700.00	325.21
100	20	78	44039800.00	282.30
100	84	49	102553700.00	249.15
100	28	99	70483900.00	254.27
100	2	66	9976800.00	755.81
Promedio			6096330	329.12

Como podemos ver de los gráficos y la tabla suministrada, al dividir los tiempos por $n * K_m * K_b$, tienden a un número constante mayor a cero. Entonces nuestro algoritmo tendría complejidad $\mathcal{O}(c * n * K_m * K_b)$, donde c es la constante a la cual converge el gráfico. Por lo tanto concluimos que los gráficos se condicen con nuestra predicción de complejidad.

1.5.2. Experimentación con instancias particulares

Queda considerar la existencia o no de instancias particulares que afecten la ejecución del algoritmo, es decir si para una determinada combinación de valores de entrada cambia la complejidad del mismo.

A partir de un análisis del algoritmo, queda evidenciado que estas instancias no existen, ya que los ciclos que determinan la complejidad del mismo se ejecutan sin importar los valores particulares en los costos de cada vehículo por etapa. Por lo tanto, se concluye que no hay instancias particulares de mejor o peor caso para este algoritmo.

2. Ejercicio 2 - Zombieland II

2.1. Problema a resolver

El problema a resolver consiste en salvar a un científico en una ciudad infestada de zombies. Para hacerlo, debemos recorrer la ciudad con una cantidad de soldados iniciales y llegar a un bunker militar. Estos soldados pelean con los zombies que hay en cada calle y mueren según la cantidad de zombies que haya (si hay mas zombies, mueren tantos soldados como el excedente entre la cantidad de zombies y soldados). Queremos que la cantidad de soldados que llegan al bunker sea la máxima posible.

La ciudad en cuestión consiste en calles horizontales y verticales. Cada calle tiene una cantidad variable de zombies.

Tenemos los siguientes parámetros de entrada:

- n = calles horizontales
- m = calles verticales
- s = soldados iniciales
- pi = esquina inicial fila
- pj = esquina inicial columna
- bi = esquina bunker fila
- bj = esquina bunker columna
- M = matriz que le asigna una cantidad z de zombies a cada calle

Donde M es una matriz de la pinta:

- $$M = \begin{pmatrix} & ch_{1,1} & \dots & ch_{1,m-1} & \\ cv_{1,1} & & cv_{1,2} & \dots & cv_{1,m} \\ & & \dots & & \\ cv_{n-1,1} & & cv_{n-1,2} & \dots & cv_{n-1,m} \\ & ch_{n,1} & \dots & ch_{n,m-1} & \end{pmatrix}$$

Con:

- $ch_{i,j}$ = #{ zombies en la calle horizontal i,j } = #{ zombies entre las esquinas $v_{i,j}$ y $v_{i,j+1}$ }
- $cv_{i,j}$ = #{ zombies en la calle vertical i,j } = #{ zombies entre las esquinas $v_{i,j}$ y $v_{i+1,j}$ }

Se atraviesa una calle sin bajas si la cantidad de soldados (s) es mayor o igual a z. Cuando z es mayor, los soldados sobrevivientes depende del calculo z - s. Si esa operación arroja un número mayor o igual a s, no hay sobrevivientes, lo que quiere decir que ese camino es inviable.

Por otro lado, queremos resolver este problema de forma que el algoritmo tenga una complejidad de $\mathcal{O}(s * n * m)$. Esto quiere decir que no podemos considerar todos los caminos del punto inicial al bunker. Si ese fuera el caso, la complejidad seria mucho mayor (ver seccion Complejidad). Por eso, es importante considerar la cantidad de soldados al buscar el mejor camino.

2.2. Resolución planteada

2.2.1. Idea y representacion de datos

Dado que el problema consiste en encontrar un camino entre las esquinas de un mapa con forma de grilla y un costo/zombies en cada calle, parece natural representar el mapa de la ciudad con un grafo.

A su vez, el grafo lo vamos a representar como una matriz de vertices. Cada vertice $v_{i,j}$ corresponde a la esquina del mapa (i,j). Estos tienen la siguiente estructura:

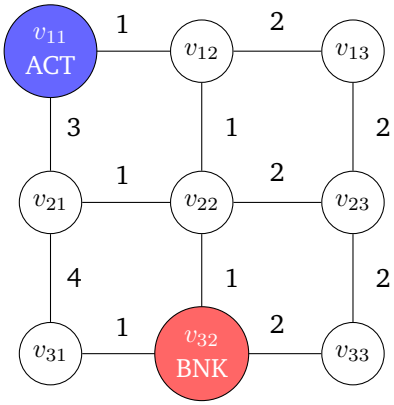
- sol_max = maxima cantidad actual de soldados que pueden llegar vivos al vertice
- left = costo de ir al vertice de la izquierda, si esto no es posible, un valor que represente a infinito
- right = costo de ir al vertice de la derecha, si esto no es posible, un valor que represente a infinito
- up = costo de ir al vertice de arriba, si esto no es posible, un valor que represente a infinito
- down = costo de ir al vertice de abajo, si esto no es posible, un valor que represente a infinito
- pred = vertice desde el cual llegue la ultima vez que actualice sol_max

Es importante aclarar que como la ciudad es una grilla, cada esquina puede tener a lo sumo cuatro esquinas adyacentes. Entonces me basta guardarme en cada nodo el costo de ir hacia cada una de ellas para no perder ningun arista del grafo representado. A su vez, esto me va a facilitar mucho la manera en la que recorro el grafo, porque toda la informacion que voy a necesitar esta en un mismo lugar.

Por ejemplo, sean:

- $n = 3$
- $m = 3$
- $s = 6$
- $pi = 1$
- $pj = 1$
- $bi = 3$
- $bj = 2$
- $M = \begin{pmatrix} 1 & 2 & \\ 3 & 1 & 2 \\ & 1 & 2 \\ 4 & 1 & 2 \\ & 1 & 2 \end{pmatrix}$

Lo comprenderiamos como:



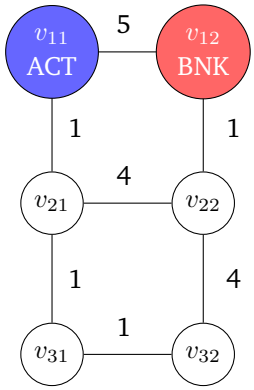
Donde BNK es la posición del bunker y ACT es la posicion actual, que al principio es donde sale el cientifico con los soldados.

Dada toda esta estructura, el algoritmo procede de la forma siguiente: Primero, rellena la matriz de vertices con los parametros de entrada. Luego, empezamos a analizar los nodos desde el punto inicial y, mientras el punto analizado p no sea el bunker, calculamos con que cantidad de soldados podemos arribar a los vértices adyacentes (la cuenta correspondiente fue explicada en la seccion previa). **Solo si** esa cantidad es superior a la maxima cantidad con la que arribamos previamente (dato que contiene el nodo), actualizamos la información del vértice adyacente con la nueva cantidad máxima de soldados, el vértice desde donde se arribo con esa cantidad y pasamos a analizar de forma recursiva el vértice adyacente. Una vez terminado esto analizamos, si es necesario, los demas vecinos de p tambien de forma recursiva. De esta forma recorremos los nodos mediante Backtracking, pero facilitado por el hecho de que solo visitamos los nodos cuando se puede llegar a estos con mas soldados que antes. Por eso, la cantidad de veces que se visita cada nodo esta acotada por la cantidad de soldados iniciales (esto lo vemos bien en la seccion de Complejidad).

Por ejemplo, dados los parametros de entrada:

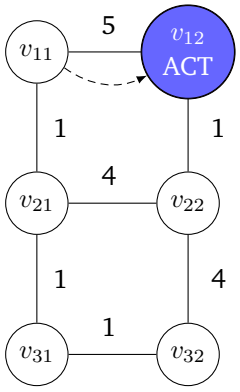
- $n = 3$
- $m = 2$
- $s = 3$
- $pi = 1$
- $pj = 1$
- $bi = 1$
- $bj = 2$
- $M = \begin{pmatrix} 5 & & \\ 1 & & 1 \\ & 4 & \\ 1 & & 4 \\ & 1 & \end{pmatrix}$

Sea $v_{i,j}$ = tupla(int sol_max, tupla(int a, int b) pred) la tupla que usamos para ver el estado de cada nodo en las distintas iteraciones del algoritmo. Si usamos flechas punteadas para representar desde cual esquina llegue a analizar la actual, el algoritmo procederia asi:



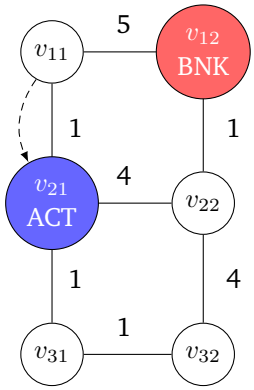
- $v_{1,1} = (3, (\infty, \infty))$
- $v_{1,2} = (0, (\infty, \infty))$
- $v_{2,1} = (0, (\infty, \infty))$
- $v_{2,2} = (0, (\infty, \infty))$
- $v_{3,1} = (0, (\infty, \infty))$
- $v_{3,2} = (0, (\infty, \infty))$

Aca vemos que desde el nodo $v_{1,1}$ se puede llegar a $v_{1,2}$ y $v_{2,1}$ con mas soldados que antes. El algoritmo entonces hace una llamada recursiva a $v_{1,2}$ con la nueva cantidad de soldados y el nuevo predecesor. Despues de resolver aquel subproblema, se fija si todavia se puede llegar a $v_{2,1}$ con mas soldados que todo lo previamente calculado. El proximo grafico parte de la llamada recursiva a $v_{1,2}$.



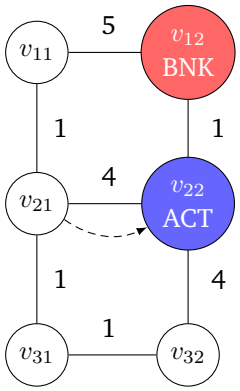
- $v_{1,1} = (3, (\infty, \infty))$
- $v_{1,2} = (1, (1,1))$
- $v_{2,1} = (0, (\infty, \infty))$
- $v_{2,2} = (0, (\infty, \infty))$
- $v_{3,1} = (0, (\infty, \infty))$
- $v_{3,2} = (0, (\infty, \infty))$

Se podria llegar con mas soldados a $v_{2,2}$, pero como $v_{1,2}$ es el bunker, dejamos de buscar (ver observacion mas abajo). Entonces hacemos backtrack a $v_{1,1}$, desde ahi podemos llegar con mas soldados a $v_{2,1}$, asi que procedemos a ese nodo:



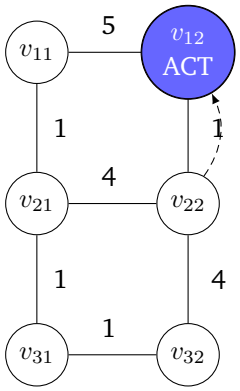
- $v_{1,1} = (3, (\infty, \infty))$
- $v_{1,2} = (1, (1,1))$
- $v_{2,1} = (3, (1,1))$
- $v_{2,2} = (0, (\infty, \infty))$
- $v_{3,1} = (0, (\infty, \infty))$
- $v_{3,2} = (0, (\infty, \infty))$

Vemos que se puede llegar a $v_{2,2}$ con mas soldados que antes.



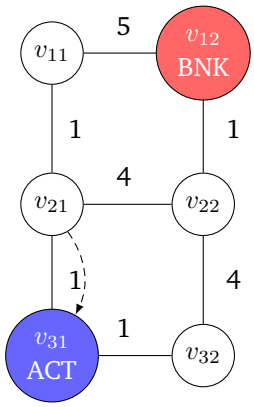
- $v_{1,1} = (3, (\infty, \infty))$
- $v_{1,2} = (1, (1,1))$
- $v_{2,1} = (3, (1,1))$
- $v_{2,2} = (2, (2,1))$
- $v_{3,1} = (0, (\infty, \infty))$
- $v_{3,2} = (0, (\infty, \infty))$

Ahora desde $v_{2,2}$ vamos a proceder a $v_{1,2}$, pero solo porque llegamos con mas soldados que antes. Cuando llegamos al nodo, actualizamos los soldados maximos y el predecesor:



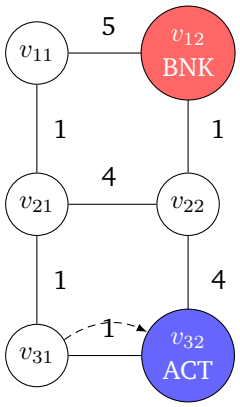
- $v_{1,1} = (3, (\infty, \infty))$
- $v_{1,2} = (2, (2,2))$
- $v_{2,1} = (3, (1,1))$
- $v_{2,2} = (2, (2,1))$
- $v_{3,1} = (0, (\infty, \infty))$
- $v_{3,2} = (0, (\infty, \infty))$

Despues, hacemos backtrack como antes y desde $v_{2,1}$ vamos a $v_{3,1}$ con mas soldados que antes:



- $v_{1,1} = (3, (\infty, \infty))$
- $v_{1,2} = (2, (2,2))$
- $v_{2,1} = (3, (1,1))$
- $v_{2,2} = (2, (2,1))$
- $v_{3,1} = (3, (2,1))$
- $v_{3,2} = (0, (\infty, \infty))$

Luego procedemos al nodo/esquina $v_{3,2}$:



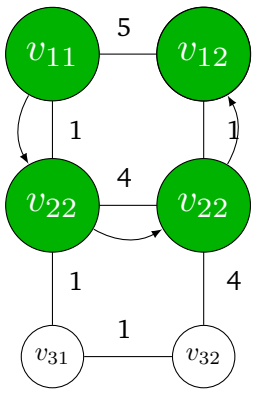
- $v_{1,1} = (3, (\infty, \infty))$
- $v_{1,2} = (2, (2,2))$
- $v_{2,1} = (3, (1,1))$
- $v_{2,2} = (2, (2,1))$
- $v_{3,1} = (3, (2,1))$
- $v_{3,2} = (2, (3,1))$

Ahora desde $v_{3,2}$ no vamos a proceder a $v_{2,2}$ ni ningun otro nodo porque llegariamos con menor o igual cantidad de soldados, esta es la **clave** para entender la complejidad. Finalmente, como el nodo $v_{3,2}$ no puede llegar a otro nodo con mas soldados que antes, hacemos backtrack. Como en el backtrack vemos que no podemos llegar a ningun otro nodo con mas soldados que el estado actual, termina la parte

principal del algoritmo (solo faltaria reconstruir el camino).

Observacion: una vez que llego al bunker, dejo de considerar caminos, es decir, no voy a tratar de llegar a ninguna otra esquina desde el bunker. Esto no nos trae ningun problema porque no existe un camino desde el punto de partida hacia el bunker que pase dos veces por el bunker y sea mejor al sub-camino desde el punto de partida hasta la primera vez que pase por el bunker.

De esta forma, reconstruir el camino es facil, dado que tenemos los datos del predecesor de cada nodo y sabemos donde esta el bunker y la posicion inicial (son parametros). Todo lo que habria que hacer es ir recorriendo desde el bunker mediante el predecesor y asi hasta el vertice inicial, agregando los vertices a una pila (asi quedan ordenados del primero al ultimo). Este es el camino que se forma en el ejemplo anterior:



- $v_{1,1}$ = (3, (∞, ∞)) = BNK.pred.pred.pred
- $v_{1,2}$ = (2, (2,2)) = BNK
- $v_{2,1}$ = (3, (1,1)) = BNK.pred.pred
- $v_{2,2}$ = (2, (2,1)) = BNK.pred
- $v_{3,1}$ = (3, (2,1))
- $v_{3,2}$ = (2, (3,1))

Observacion: si no se puede construir un camino, el bunker va a tener como maxima cantidad de soldados 0 y como predecesor (∞, ∞). Por eso detectar este caso es simple y no nos produce ningun problema.

Como vemos, estamos abarcando de esta manera todos los vértices a los que es posible llegar con los soldados que disponemos en cada caso hasta el bunker. Como vamos a ver en la seccion de Complejidad, visitar los vecinos de un nodo solo cuando se pueda llegar con mas soldados que antes, nos lleva a analizar cada vértice a lo sumo s veces.

2.2.2. Pseudocodigo

```
Sea Esquina una tupla(sol_max:int, left:int, right:int,
        up:int, down:int, predecesor:(int, int))
  con left,...,down la cantidad de zombies entre la esquina y su vecina
  con sol_max la maxima cantidad de zombies actual que pueden llegar a la esquina
  con predecesor la esquina desde donde llegue la ultima vez que actualice sol_max
Sea Mapa un vector(vector(Esquina))
Sea n = calles horizontales
Sea m = calles verticales
Sea s = soldados iniciales
Sea pi = esquina inicial fila
Sea pj = esquina inicial columna
Sea bi = esquina bunker fila
Sea bj = esquina bunker columna
Sea M = matriz que le asigna una cantidad $z$ de zombies a cada calle

Parsear Entrada(n, m, s, pi, pj, bi, bj, M):
  Creamos un mapa llamado map de tamano n*m que es una matriz de vertices

  Inicializamos los vertices de map con los valores de los zombis de cada calle,
  segun corresponda en los campos left, right, up y down,
  sol_max en 0 y predecesor en (INFINITO, INFINITO).
  En caso de no existir un camino a la left, ponemos INFINITO,
  lo mismo para el resto de las posiciones.

  En el campo sol_max de map[pi][pj], ponemos s.

  Luego llamamos la función auxiliar Resolver, con los parámetros mapa,
  posición inicial, posición bunker y, como posicion actual, la inicial.

  Reconstruimos el camino hacia el bunker (si es que existe), es decir, vamos
  desde el bunker recorriendo las esquinas mediante el campo predecesor hasta
  llegar a la posicion inicial y agregando las esquinas a una pila. Finalmente,
  mostramos la solución recorriendo los valores de la pila.

Resolver(map, pi, pj, bi, bj, actuali, actualj):
  Si el vertice <actuali, actualj> es igual a <bi, bj> (el actual es el bunker)
    hago un return
  En caso contrario
    Para cada vecino vcn de map[actuali][actualj] con
        costo (left..down) segun corresponda
    Si puedo ir desde el nodo actual hasta vcn (el costo no es infinito)
    Calculo la cantidad de soldados que llegarian vivos hasta esa esquina
      comparando map[actuali][actualj].sol_max con los zombies en la calle respectiva
      si hay mas soldados o igual cantidad, no muere ninguno
      si hay mas zombies, mueren tantos soldados como el excedente
          entre los zombies y los soldados
    Si ademas puedo llegar hasta vcn con mas soldados que todo lo previamente calculado
      actualizo el campo sol_max de vcn con los sobrevivientes
      actualizo el campo predecesor con la posicion actual
      hago una llamada recursiva a Resolver(map, pi, pj, bi, bj, vcn.fila, vcn.columna)
```

2.3. Complejidad propuesta

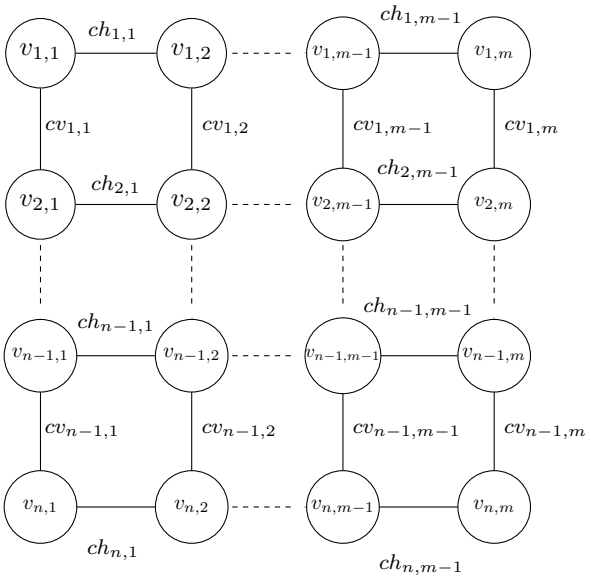
Dados los parámetros de entrada:

- n = calles horizontales
- m = calles verticales
- s = soldados iniciales
- pi = esquina inicial fila
- pj = esquina inicial columna
- bi = esquina bunker fila
- bj = esquina bunker columna

Y la matriz de zombies

- $M = \begin{pmatrix} & ch_{1,1} & ... & ch_{1,m-1} & \\ cv_{1,1} & & cv_{1,2} & ... & cv_{1,m} \\ & & ... & & \\ cv_{n-1,1} & & cv_{n-1,2} & ... & cv_{n-1,m} \\ & ch_{n,1} & ... & ch_{n,m-1} & \end{pmatrix}$

Construimos una matriz de vertices de la forma:



Como vemos, si en total hay n calles horizontales y m verticales, el total de nodos seria $n * m$. Si representara el grafo como una matriz de adyacencia, solo el costo de cargarla tendria una complejidad mayor a $\mathcal{O}(n * m)$. En cambio, con la forma elegida, solo hay que guardar una cantidad constante de informacion en cada uno de los $n * m$ nodos del grafo.

No podemos considerar todos los caminos del punto inicial al bunker. Si ese fuera el caso, la complejidad seria mucho mayor, ya que la complejidad mencionada recorre cada esquina de la ciudad a lo sumo s veces. Considerando todos los posibles caminos la complejidad seria independiente de s, pero la cantidad de veces que analizamos cada una de las $n * m$ esquinas no seria constante. Por ejemplo, si representaramos las $n * m$ esquinas como vertices y las calles como aristas de un grafo y usamos alguna variacion de un algoritmo para hallar caminos minimos como el de Dijkstra, entonces la complejidad quedaria $\mathcal{O}(n^2 * m^2)$. Aun optimizandolo con una cola de prioridad la complejidad seguiria siendo $\omega(n * m)$.

Como explicamos anteriormente el algoritmo empieza a analizar los nodos desde el punto inicial y, mientras el punto analizado p no sea el bunker, calcula con que cantidad de soldados podemos arribar a los vértices adyacentes. La forma en que recorre los nodos es parecida a Backtracking, prioriza la profundidad del camino, pero dada una esquina, solo hace una llamada recursiva a un nodo vecino cuando podemos llegar a el con mas soldados que todos los caminos antes calculados. Lo importante de este procedimiento es el hecho de que cada uno de los $n * m$ nodos es analizado cuando se puede llegar al nodo con una cantidad de soldados mayor a la previamente calculada. Es decir, si yo llegue a un nodo con k soldados, ese nodo no se volvera a analizar con una cantidad de soldados menor o igual a k. Entonces, como uno empieza el algoritmo con s soldados y la cantidad de soldados no puede aumentar (solo disminuir o quedar fija), a lo sumo se visita cada nodo con 1, 2, ... , s soldados en ese orden. Entonces, a diferencia de un Backtracking normal, cada nodo puede ser visitado a lo sumo s veces.

Aparte de lo ya mencionado, el algoritmo tambien tiene que reconstruir el camino de la posicion inicial al bunker (si es que existe) e imprimirlo. De todas maneras, esto se resuelve facilmente recorriendo las esquinas desde el bunker hasta el punto inicial con el campo de predecesor que cada una contiene. Mientras visitamos los vertices, los vamos guardando en una pila (pues los queremos imprimir en el orden inverso) y luego los imprimimos. Como a lo sumo pasamos una vez por cada uno de los vertices y hay $n * m$ de ellos, este paso tiene complejidad $\mathcal{O}(n * m)$.

Por lo tanto, vimos que el grafo en cuestion tiene $n * m$ nodos y cargarlo tanto como imprimirlo al final cuesta $\mathcal{O}(n * m)$. Ademas, como cada nodo tiene a lo sumo cuatro vecinos y por cada uno de ellos el algoritmo solo hace comparaciones para saber si hay que analizarlo, podemos decir que se realiza trabajo constante (ademas de las llamadas recursivas) en cada paso de la recursion. Entonces dado que el algoritmo pasa por cada nodo a lo sumo s veces, son $n * m$ nodos y realiza trabajo constante en cada uno, concluimos que la complejidad del algoritmo es $\mathcal{O}(s * n * m)$.

2.4. Implementación en C++

```
#include <vector>
#include <iostream>
#include <stack>
using namespace std;

#define INFINITO -1

struct Esquina{
    int sol_max;
    int left;
    int right;
    int up;
    int down;
    pair<int, int> pred;

    Esquina() : sol_max(0), left(INFINITO), right(INFINITO), up(INFINITO
       ), down(INFINITO), pred(make_pair(INFINITO,INFINITO)){};
};

typedef vector< vector < Esquina > > Mapa;

void resolver(Mapa& mapa, pair<int, int> pos_actual, pair<int, int>
   bunker);
void imprimir_resultado(Mapa& mapa, pair<int, int> pos_inicial, pair<int
   , int> bunker);
void mostrar(Mapa& m); //debug
int sobrevivientes(int soldados, int zombies);

// Implementacion. Contiene el cargado de input mas la resolucion del
   ejercicio.
int main() {
    int n, m, s;
    cin >> n;
    cin >> m;
    cin >> s;

    pair<int, int> pos_inicial, bunker;
    cin >> pos_inicial.first;
    cin >> pos_inicial.second;
    cin >> bunker.first;
    cin >> bunker.second;

    Mapa mapa = Mapa(n, vector< Esquina >(m, Esquina()));

    for (int i = 0; i < n; i++) {
        for (int j = 1; j < m; j++){
            int c;
            cin >> c;
            mapa[i][j-1].right = c;
            mapa[i][j].left = c;
        }
        if (i < n-1){
```

```
            for (int j = 0; j < m; j++){
                int c;
                cin >> c;
                mapa[i][j].down = c;
                mapa[i+1][j].up = c;
            }
        }
    }

    mapa[pos_inicial.first][pos_inicial.second].sol_max = s;

    resolver(mapa, pos_inicial, bunker);

    imprimir_resultado(mapa, pos_inicial, bunker);
    cout << endl;

    return 0;
}

void imprimir_resultado(Mapa& mapa, pair<int, int> pos_inicial, pair<int
    , int> bunker){
    cout << mapa[bunker.first][bunker.second].sol_max << endl;
    if (mapa[bunker.first][bunker.second].sol_max > 0){
        stack< pair<int,int> > esquinas;
        pair<int,int> actual = bunker;
        while (actual != pos_inicial){
            esquinas.push(actual);
            actual = mapa[actual.first][actual.second].pred;
        }
        esquinas.push(pos_inicial);
        while (esquinas.size() > 0){
            cout << esquinas.top().first << "␣" << esquinas.top().second
                << endl;
            esquinas.pop();
        }
    }
}

void mostrar(Mapa& m) {
    cout << endl;
    int n = m.size();
    for(int i = 0; i < n; ++i) {
        int t = m[i].size();
        for (int j = 0; j < t; j++) {
            if (m[i][j].sol_max == 0)
                cout << "X␣";
            else
                cout << m[i][j].sol_max << "␣";
        }
        cout << endl;
    }
}

int sobrevivientes(int soldados, int zombies){
```

```
    return soldados < zombies ? max(2*soldados-zombies, 0) : soldados;
}

void resolver(Mapa& mapa, pair<int, int> pos_actual, pair<int, int>
    bunker){
    if (pos_actual != bunker){
        int i = pos_actual.first;
        int j = pos_actual.second;
        Esquina actual = mapa[i][j];
        if (actual.left != INFINITO && sobrevivientes(actual.sol_max,
            actual.left) > mapa[i][j-1].sol_max){
            mapa[i][j-1].sol_max = sobrevivientes(actual.sol_max, actual
                .left);
            mapa[i][j-1].pred = make_pair(i,j);
            resolver(mapa, make_pair(i,j-1), bunker);
        }
        if (actual.right != INFINITO && sobrevivientes(actual.sol_max,
            actual.right) > mapa[i][j+1].sol_max){
            mapa[i][j+1].sol_max = sobrevivientes(actual.sol_max, actual
                .right);
            mapa[i][j+1].pred = make_pair(i,j);
            resolver(mapa, make_pair(i,j+1), bunker);
        }
        if (actual.up != INFINITO && sobrevivientes(actual.sol_max,
            actual.up) > mapa[i-1][j].sol_max){
            mapa[i-1][j].sol_max = sobrevivientes(actual.sol_max, actual
                .up);
            mapa[i-1][j].pred = make_pair(i,j);
            resolver(mapa, make_pair(i-1,j), bunker);
        }
        if (actual.down != INFINITO && sobrevivientes(actual.sol_max,
            actual.down) > mapa[i+1][j].sol_max){
            mapa[i+1][j].sol_max = sobrevivientes(actual.sol_max, actual
                .down);
            mapa[i+1][j].pred = make_pair(i,j);
            resolver(mapa, make_pair(i+1,j), bunker);
        }
    }
}
```

2.5. Experimentación computacional

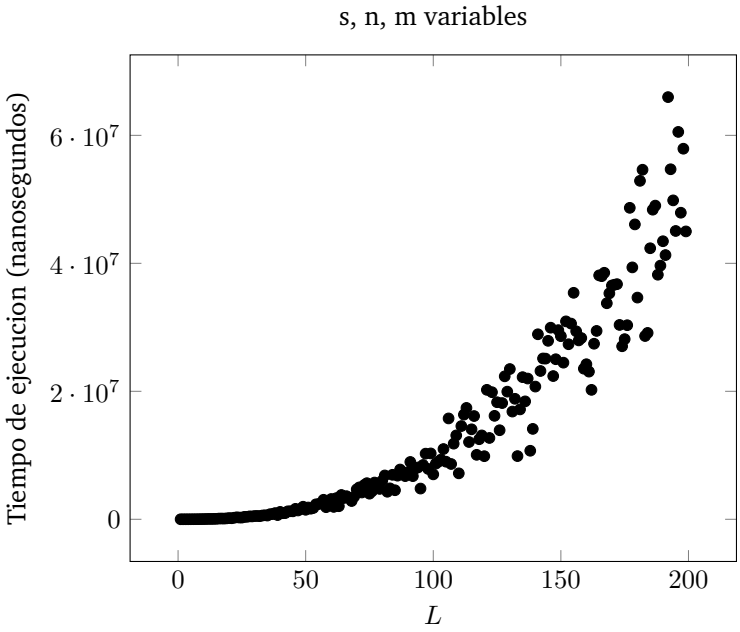
Consideraciones:

- Todos los experimentos fueron hechos bajo las mismas condiciones (computadora, procesos abiertos, alimentacion, etc..)
- Para cada medicion exactamente igual se tomaron 20 pruebas, los valores que aparecen en los graficos son los del promedio entre las diferentes corridas.
- Se midieron los tiempos con la biblioteca chrono y estos fueron convertidos a nanosegundos.
- Los valores aleatorios que fuimos tomando en algunos de los experimentos fueron tomados con una distribucion uniforme para el rango requerido.
- Tomamos entre 100 y 200 valores de entrada diferente en cada experimento. Por ejemplo, si el tamano de entrada va de 1 a 5000, mediriamos los valores con tamano de entrada 1, 50, 100, 150, ..., 5000. Esto sirve para no sobrepopular los graficos.
- Salvo indicado lo contrario, la posicion inicial del cientifico y la posicion del bunker van a ser aleatorias (si bien en un rango valido).
- Vamos a querer medir la rapidez del algoritmo segun los parametros:
  - n = calles horizontales
  - m = calles verticales
  - s = soldados iniciales
  - y la cantidad de zombies en cada esquina
- Vamos a querer mostrar que el algoritmo funciona como explicamos y que el trabajo que realiza se condice con nuestra cota analizada en la seccion de Complejidad. Nuestra cota fue $\mathcal{O}(s * n * m)$.

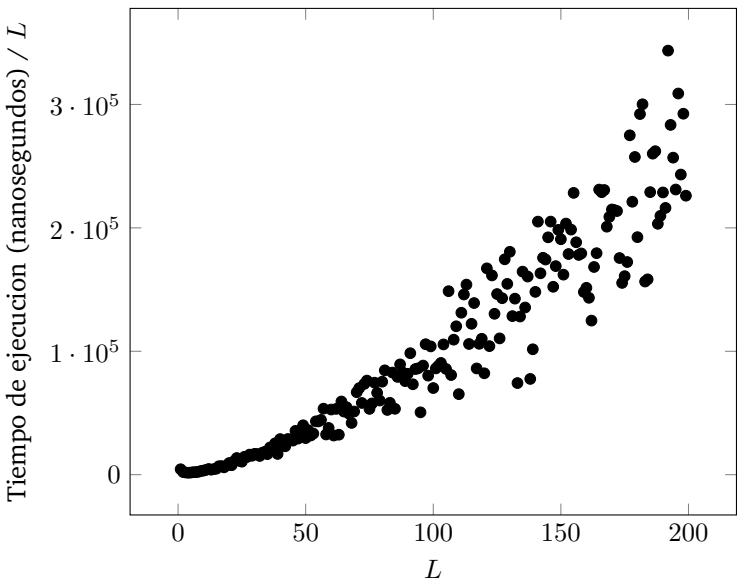
2.5.1. Experimentación con instancias aleatorias

En esta seccion, realizamos una serie de experimentos donde la cantidad de zombies en cada calle es aleatoria. Para probar la eficiencia del algoritmo vamos fijando algunos parametros de entrada y variando los otros. Si bien la cantidad de zombies va a ser aleatoria, su rango de valores posibles va a estar entre 0 y $2 * s$ (dos veces la cantidad de soldados iniciales). Esto es importante y necesario pues si no lo hicieramos asi, los soldados serian destruidos por los zombies muy rapidamente, ya que habria una cantidad de zombies enorme en cada calle en comparacion a los soldados.

Experimento 1 Para el primer experimento, vamos a variar los parametros de entrada s, n y m uniformemente. Es decir, el tamano de entrada es una variable L donde $s = n = m = L$ que se va incrementando. Los valores de L van a variar entre 1 y 200. Vamos a querer ver que nuestra cota de complejidad es $\mathcal{O}(L^3) = \mathcal{O}(L * L * L) = \mathcal{O}(s * n * m)$. Veamos los resultados:

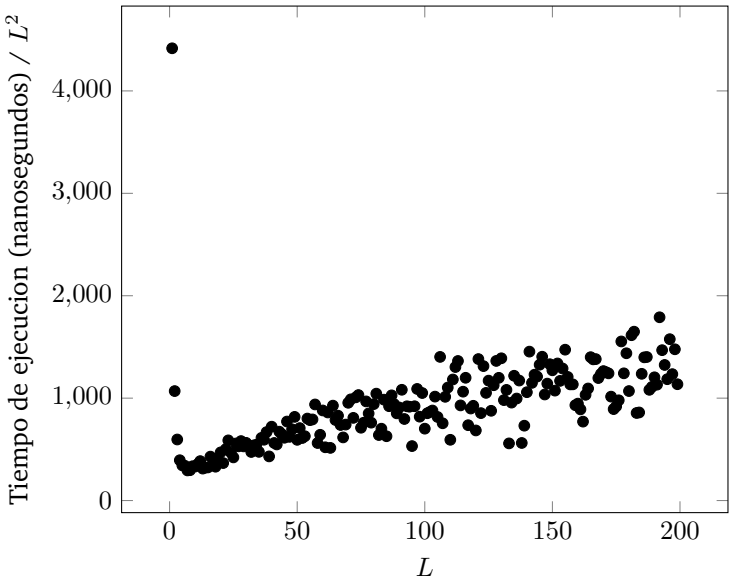


El grafico tiene cierta pendiente que es claramente no lineal. Queremos ver que es una pendiente como la de una funcion $f(L) = L^3$.Si dividimos los tiempos por L, tenemos el siguiente grafico:



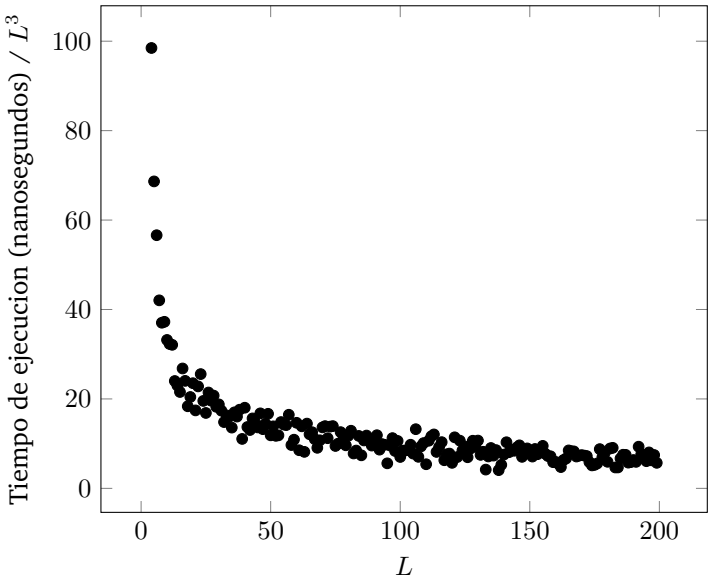
Dividiendo por segunda vez los tiempos por L nos queda:

s, n, m variables, divido los tiempos originales por L^2



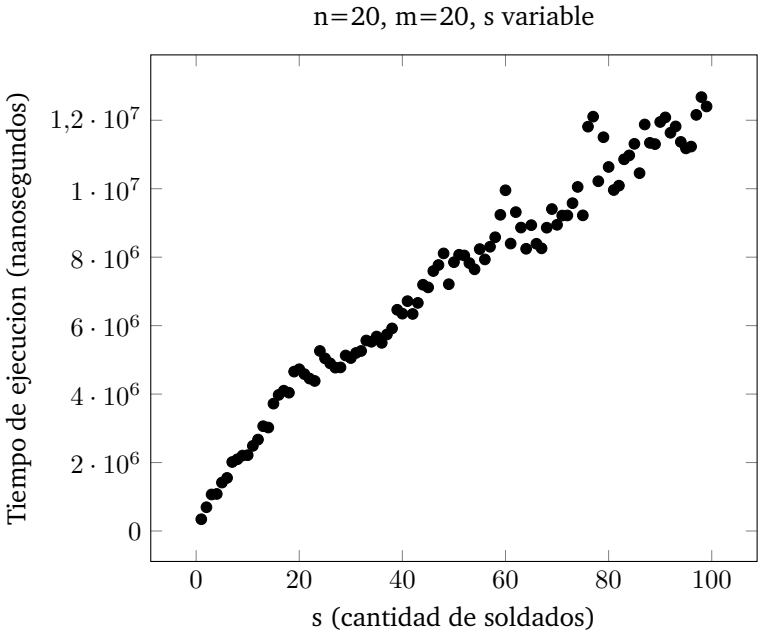
Esto ultimo parece ser lineal, lo que seria un indicio que estabamos en lo correcto y la complejidad del algoritmo sigue la curva de una funcion tal como $f(L) = L^3$. Veamos el grafico que queda si dividimos los tiempos por L una vez mas:

s, n, m variables, divido los tiempos originales por L^3

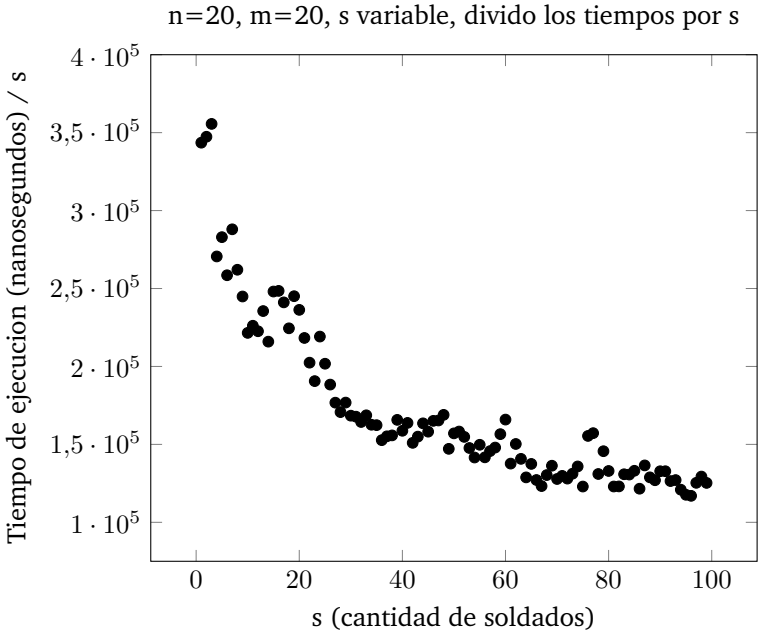


La pendiente tiende a una constante por arriba del 0. Esto quiere decir que estabamos en lo correcto, ya que la funcion sigue una curva al estilo cubico. Entonces se condice con nuestra cota de complejidad $\mathcal{O}(s * n * m)$, pues fijamos $s = n = m = L$ y nos quedo un grafico que sigue la curva de una funcion polinomial de grado 3.

Experimento 2 Para este experimento, fijamos los valores de n y m en 20 mientras variamos los valores de s, los resultados fueron:

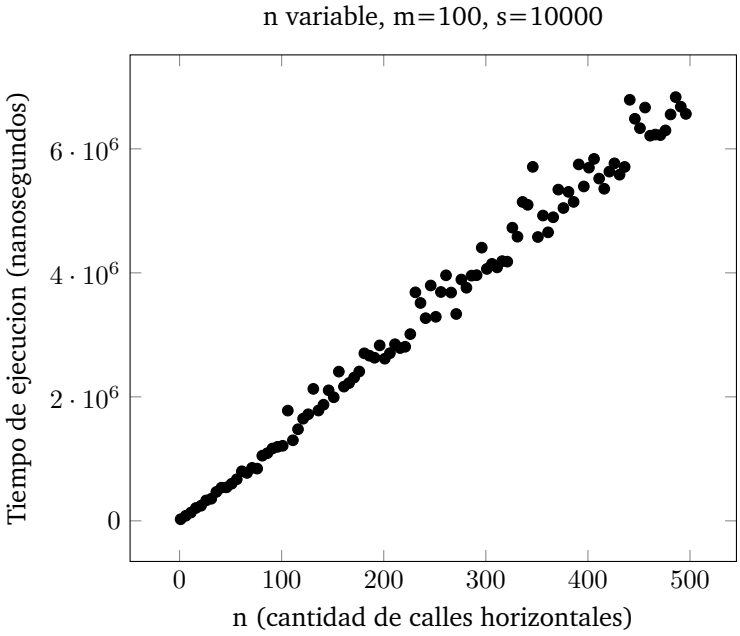


Esto pareceria ser lineal desde la segunda mitad del dominio, veamos que pasa cuando dividimos los tiempos por el tamano de entrada s:

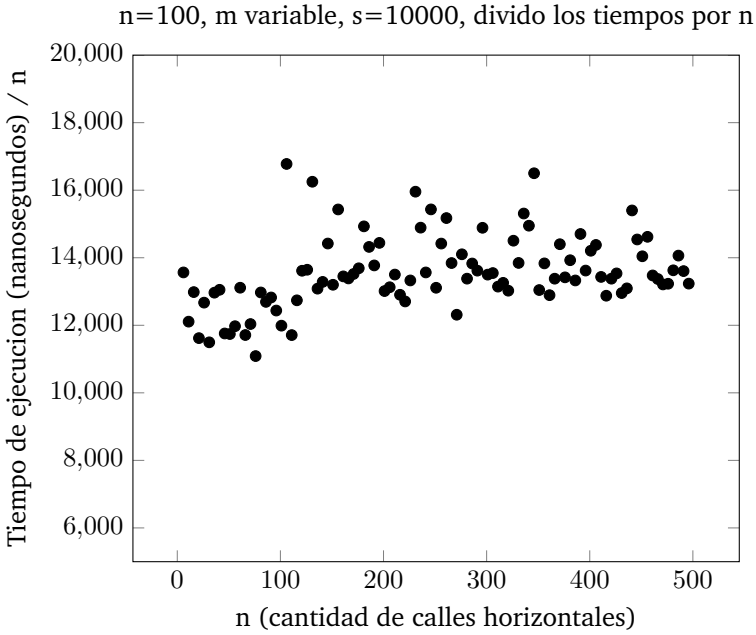


Parece que la pendiente se vuelve constante y la constante a la que tiende esta por arriba del 0. Esto se condice con nuestra cota de complejidad $\mathcal{O}(s * n * m)$, pues fijamos $n = 20, m = 20$ y el tiempo varia linealmente con respecto a s.

Experimento 3 En este experimento, fijamos los valores de m en 100 y s en 10000 mientras variamos los valores de n, los resultados fueron:

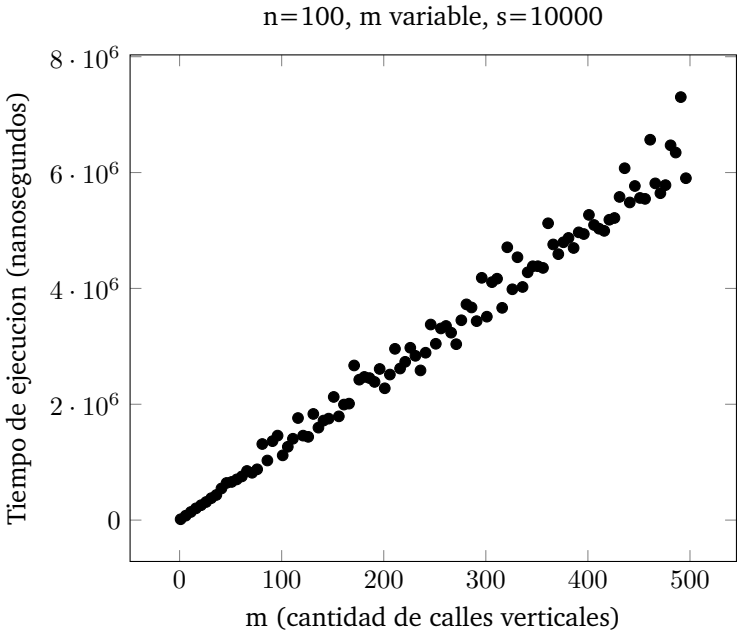


Tiene una clara pinta de lineal, veamos lo que pasa cuando dividimos los tiempos por el tamano de entrada n:

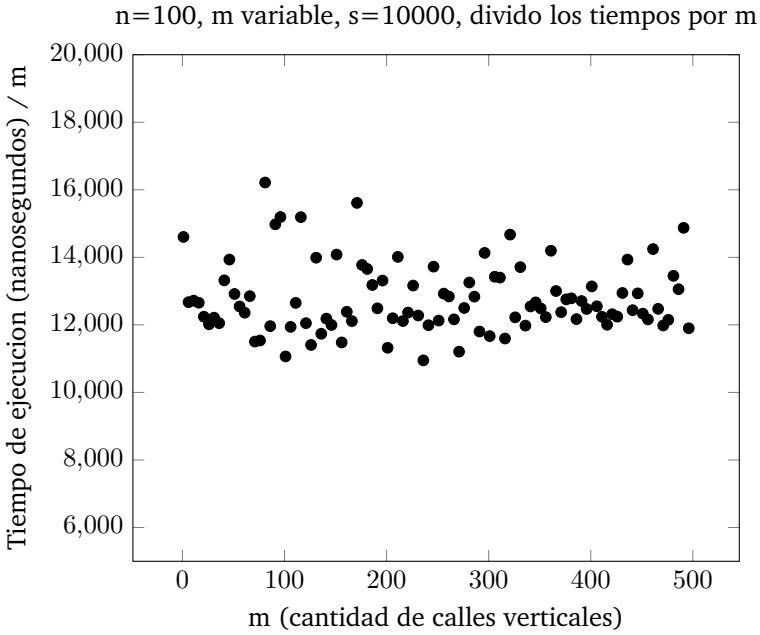


Como vemos, la pendiente tiende a una constante por arriba del 0. Esto se condice con nuestra cota de complejidad $\mathcal{O}(s * n * m)$, pues fijamos $m = 100, s = 10000$ y el tiempo varia linealmente con respecto a n.

Experimento 4 Fijamos los valores de n en 100 y s en 10000 mientras variamos los valores de m, los resultados fueron:



Es muy parecido al experimento anterior. La curva parece lineal, veamos lo que pasa cuando dividimos los tiempos por el tamano de entrada m:

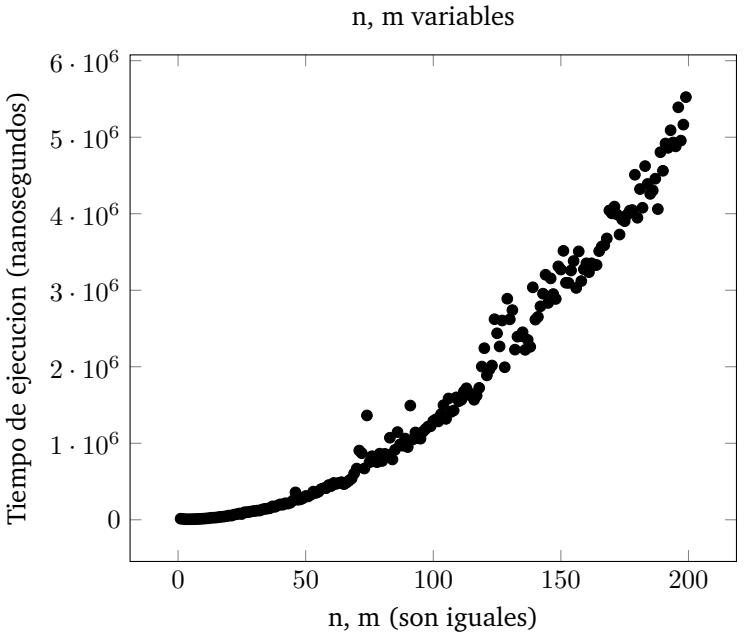


Esta pendiente tambien tiende a una constante por arriba del 0. Esto se condice con nuestra cota de complejidad $\mathcal{O}(s * n * m)$, pues fijamos $n = 100, s = 10000$ y el tiempo varia linealmente con respecto a m.

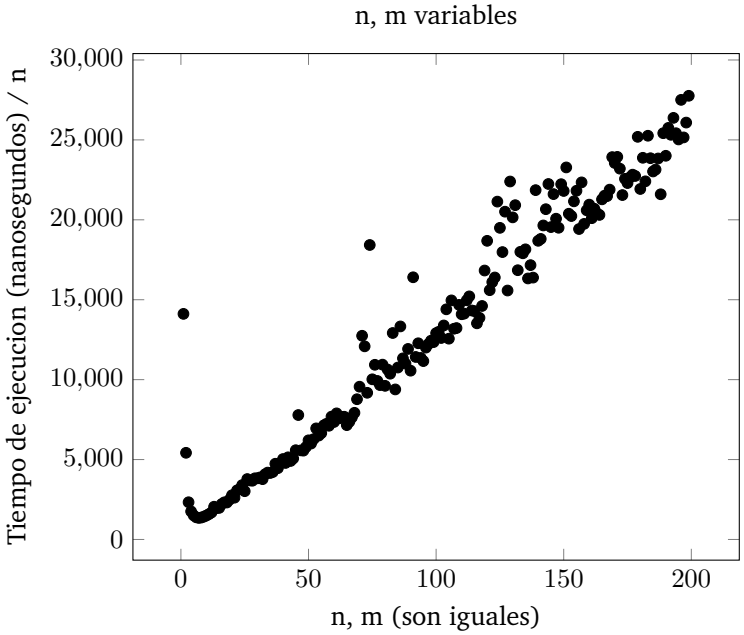
2.5.2. Experimentación con instancias particulares

Cuando analizamos instancias particulares, nos va a interesar fijar la cantidad de zombies en cada calle al contrario que las instancias aleatorias. A la vez vamos a ir variando ciertos parametros de entrada.

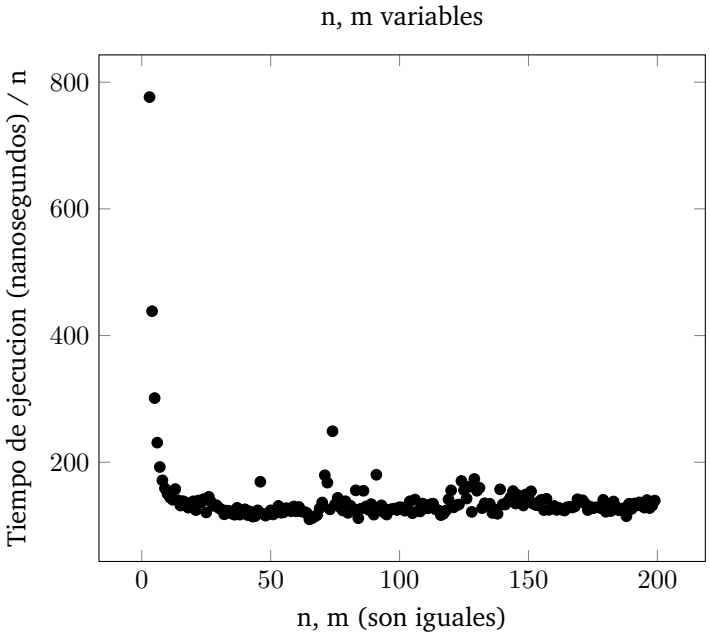
Experimento 1: Mejor caso 1 En este caso vamos a fijar la cantidad de zombies en cada calle a 0. Variamos los parametros de entrada n y m uniformemente y dejamos la cantidad de soldados fija en 1000:



Dividiendo por n nos queda:

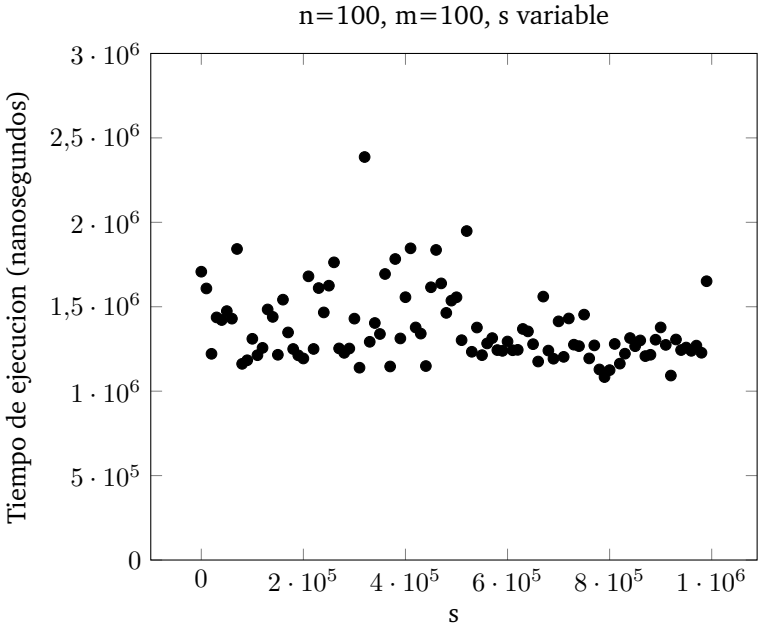


Esto es algo de pinta lineal. Si dividimos por m nos queda:



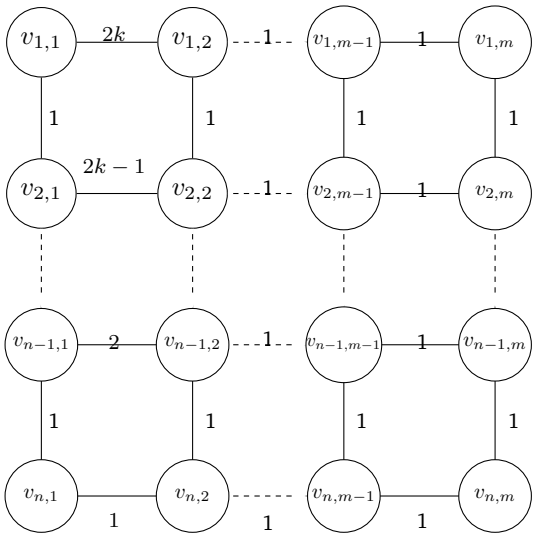
Que tiende a una constante por encima de 0. Esto se condice con nuestra cota de complejidad $\mathcal{O}(s * n * m)$, pues fijamos s y el tiempo varia linealmente con respecto a $n * m$.

Experimento 2: Mejor caso 2 Como antes, la cantidad de zombies en cada calle va a ser 0. Pero a diferencia del caso anterior, fijamos los parametros n y m en 100 y variamos la cantidad de soldados:



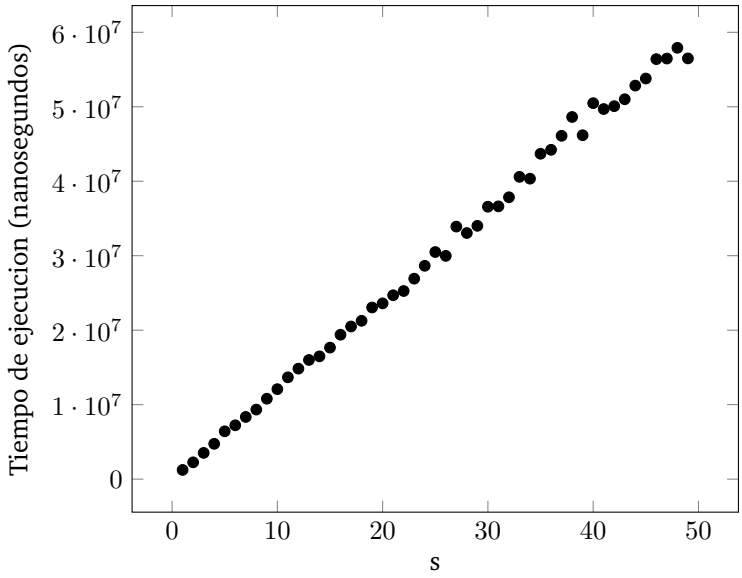
Si bien la cantidad de soldados varia en un rango muy grande, el tiempo no varia demasiado. Esto tiene sentido ya que tenemos la misma cantidad de esquinas para llegar pero siempre llegamos una vez sola. Es decir, como la cantidad de zombies en cada calle es 0, solo llego a cada esquina una vez (con la maxima cantidad de soldados posibles). Esto se condice con nuestra explicacion sobre la forma en que funciona el algoritmo.

Experimento 3: Peor caso Generar una instancia con uno de los peores casos no es simple. En este caso, vamos a generar un peor caso analizando el codigo del algoritmo. Este hace Backtracking y analiza las posibilidades en el orden: izquierda, derecha, arriba, abajo. Vamos a hacer una instancia donde maximizemos la cantidad de veces que pasamos por cada esquina. Sea k una variable, vamos a configurar $n = 2k, m = 2k$ e ir variando s entre 1 y k. Ademas vamos a tener las siguientes cantidades de zombies:

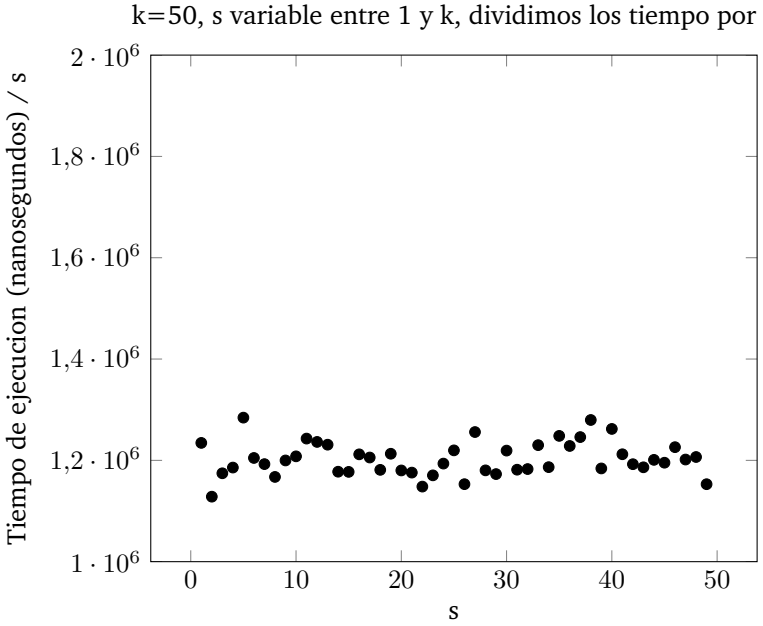


Es decir, todas las calles van a tener 1 zombie excepto la primer calle horizontal de cada fila, estas van a tener $2k, 2k-1, 2k-2, ..., 3, 2, 1$ zombies en ese orden. Ademas, vamos a fijar la posicion inicial del cientifico en (0,0) y la posicion del bunker en la esquina de abajo a la derecha, garantizando que el ejemplo funcione bien. Como el algoritmo analiza ir hacia los nodos vecinos en el orden: izquierda, derecha, arriba y abajo, va a llenar casi todos los nodos de la matriz con un 1, luego con un 2, despues con un 3 y asi hasta s. Seteando $k = 50$, los resultados fueron los siguientes:

k=50, s variable entre 1 y k



Como esto tiene pinta de lineal en s, dividimos los tiempos por s:



Y esto parece tender a una constante por encima de 0. Por lo tanto esto se condice con nuestra cota de complejidad porque aun en uno de los peores casos, el tiempo que tarda el algoritmo varia linealmente con respecto a s.

3. Ejercicio 3 - Refinando petróleo

3.1. Problema a resolver

El problema a resolver consiste en encontrar un algoritmo que dé un plan de acción para construir una cantidad de refinerías y tuberías entre pozos de petroleo con costo mínimo. Para esto se provee una cantidad de pozos de petroleo, una cantidad de tuberías que se pueden construir entre dos pozos y un costo fijo de poner una refinería en un pozo dado.
Se debe encontrar, con el menor costo posible, un plan indicando cuales son las refinerías y tuberías a construir en el que, o bien todas los pozos tengan una refinería, o bien estén conectados a una mediante una o un conjunto de tuberías.

Tenemos los siguientes parámetros de entrada:

- n = Cantidad de pozos
- m = Cantidad de tuberías posibles
- c = Costo por refinería
- Luego vienen m líneas con tuberías donde T_i representa a la tubería en la línea i con $1 \leq i \leq m$ y tiene los siguientes parámetros:
  - o_i = Tubería de orígen
  - d_i = Tubería de destino
  - c_i = Costo de la tubería

3.2. Resolución planteada

Para resolver este problema, modelaremos la entrada con un grafo, en donde cada pozo es un nodo y cada tubería que podemos construir es una arista con el costo de la construcción de la misma.

- Ejemplo 1:

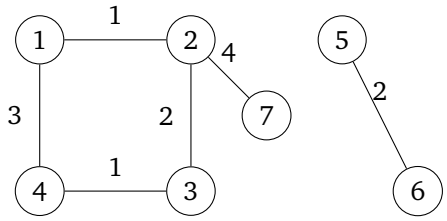
  Sean:

  - $n = 7$
  - $m = 6$
  - $c = 3$

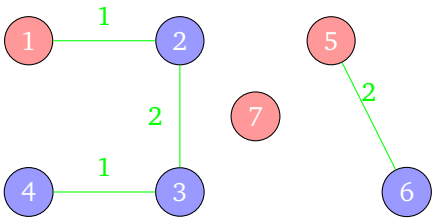
Y luego m tuberías:

T_i	o_i	d_i	c_i
T_1	1	2	1
T_2	3	4	1
T_3	2	3	2
T_4	1	4	3
T_5	5	6	2
T_6	2	7	4

Ésta entrada particular la comprenderíamos con el siguiente grafo:



Y una solución válida (no es única) para el siguiente caso se representaría con el siguiente grafo:



Donde los nodos rojos representan pozos con refinería, los ejes verdes representan las tuberías que fueron construidas y los nodos azules representan pozos que no tienen una refinería pero están conectadas a una por ejes verdes.

Ahora, observemos lo siguiente. Si yo tengo dos pozos unidos por una posible tubería y ésta tiene costo mayor al costo de poner una refinería, entonces esa tubería nunca estará construida en la solución. Por ejemplo:

- Ejemplo 2:

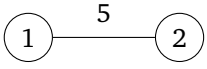
  Sean:

  - $n = 2$
  - $m = 1$
  - $c = 3$

  Y luego m tuberías:

T_i	o_i	d_i	c_i
T_1	1	2	5

  Esta entrada particular la comprenderíamos con el siguiente grafo:



  En este caso, construyendo la tubería, el grafo solución sería el siguiente:

Acá tendríamos un costo total de 8 (una refinería con costo 3 más la tubería con costo 5).

Sin embargo,la solución correcta sería la siguiente:

Donde el costo total sería 6 (dos refinerías de costo 3)

Y esto lo podemos generalizar:

Lema 3.1. *sean dos componentes conexas C_1 y C_2 con las tuberías necesarias construidas para garantizar un costo mínimo, y sea T una posible tubería entre un pozo en C_1 y uno en C_2. T solo será construída en una solución de costo mínimo si su costo es menor o igual al costo de la refinería.*

Demostración. Sea $costo()$ la función que recibe un conjunto de pozos y tuberías construidas y devuelve el costo total de la construcción.

Sea c el costo de una refinería.

Una componente conexa que está en una solución válida tiene costo mínimo, por ende, sólo tiene una refinería en ella porque en caso de tener un pozo que este conectado a una refinería por tuberías, y que a su vez tenga una refinería en el pozo, ésta no sería una solución mínima ya que puedo eliminar esa refinería reduciendo el costo total y el pozo seguiría estando conectado con una.

Ahora, sean dos componentes conexas C_1 y C_2 con las tuberías necesarias construidas para garantizar un costo mínimo, y sea T una posible tubería entre un pozo en C_1 y uno en C_2.

El costo de la tubería T es $costo(T)$.

Mi costo acumulado de C_1 y C_2 es $costo(C_1)$ y $costo(C_2)$ respectivamente.

Como estamos buscando minimizar el costo, T debe ser construida si se da lo siguiente:

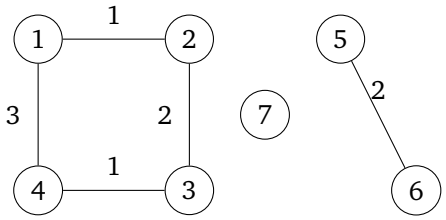
$$costo(C_1) + costo(C_2) \geq costo(C_1) + costo(C_2) + costo(T) - c$$

Entonces, despejando, solo debe ser construida si:

$$c \geq costo(T)$$

□

Por lo tanto, ya que una tubería con costo mayor al de una refinería nunca va a ser construida, podríamos no agregarla al modelar la entrada en un grafo. En el ejemplo 1, el grafo quedaría de la siguiente manera:



Entonces, de ahora en adelante modelaremos la entrada en un grafo, pero solo añadiremos como aristas las tuberías que tengan costo menor al costo de la refinería.

Notemos que podemos dividir nuestro problema en un subproblema por cada componente conexa que tenga nuestra entrada.

Necesitamos entonces, encontrar una solución mínima en cada componente conexa. Notemos ahora que si una componente conexa está en el modelo, entonces existe una solución con una sola refinería para toda la componente, ya que se puede llegar de cualquier pozo a cualquier otro mediante tuberías posibles y que todas las tuberías tienen costo menor al costo de una refinería. Notemos también que la solución no tendrá ciclos, ya que de tenerlos, ésta solución no sería mínima porque se podría eliminar

una tubería perteneciente al ciclo, reduciendo el costo total de la construcción y aún así se podría llegar de cualquier nodo de la componente a cualquier otro mediante tuberías.

Sacando en limpio estos datos, lo que estamos buscando en cada componente es encontrar una solución que recorra todos los nodos de la componente, es decir, que sea conexa, que no tenga ciclos, y que a la vez tenga costo mínimo. Entonces, lo que estamos buscando es encontrar el árbol generador mínimo de cada componente conexa.

Ahora, si encuentro un árbol generador mínimo de cada componente conexa del grafo con el que modelamos la entrada, estaría consiguiendo una solución con costo mínimo. Entonces, lo que el problema realmente nos está pidiendo es encontrar un bosque generador mínimo del grafo, ya que debemos minimizar los costos de la construcción de refinerías y tuberías.

Para resolver el siguiente problema, usaremos el algoritmo de Kruskal.
Dado un grafo, el algoritmo de Kruskal se encarga de encontrar el árbol generador mínimo en el mismo, y en caso de no ser posible porque el grafo tiene más de una componente conexa, el algoritmo encuentra el bosque generador mínimo.

El algoritmo de Kruskal, ordena primero las aristas del grafo según su costo en una estructura y luego recorre la misma en orden fijándose por cada arista si los dos nodos pertenecen a la misma componente, si estos no pertenecen a la misma componente, agrega la arista y une las componentes, y si pertenecen a la misma componente desestima esa arista ya que ya están conectadas.

De esta forma, al finalizar de recorrer la estructura quedarían tantos árboles como componentes conexas tiene el grafo, como estos árboles tienen costo mínimo, el bosque resultante es el bosque generador mínimo.

En nuestro caso, usaremos como estructura una cola de prioridad que priorice por menor costo. Además, para modelar el grafo correctamente, solo añadiremos a la cola de prioridad las tuberías posibles que tengan costo menor al costo de construir una refinería.

3.2.1. Pseudocódigo

Pseudocódigo de la solución

```
//Creamos un entero con el costo de crear una refinería en cada pozo
int costoTotal <- n*m

//Creo una cola de prioridad
ColaPrior tuberías

//Sea t el conjunto con las tuberías que se reciben en el input
//Encolamos en la cola de prioridad las tuberías de t con costo menor a c
Por cada tubería i de las m posibles que se ingresan en t
        Si el costo de la tubería i es menor a c
            tuberías.encolar(t[i])
    Fin si
Fin por

//Creamos un vector de tuberías donde guardare las tuberías que se construyen
Vector tuberíasConstruidas

//Creamos un conjunto disjunto con los pozos
UnionFind componentes(n)

Mientras tuberías no este vacío
  //Sea top el primer elemento de la colaPrior tuberías
          top <- tuberías.primero()

    Si el origen de la tuberías top no esta en el mismo conjunto que el destino
      //Unimos los conjuntos a los que pertenecen ambos pozos
        componentes.unir(top.origen,top.destino)

        //Agregamos a top al vector de tuberías construidas
        tuberíasConstruidas.agregarAtras(top)

        //Bajamos el costo total
        costoTotal <- costoTotal - c + top.costo
    Fin si

    //Desencolamos a top de la cola de prioridad
    tuberías.desencolar()
Fin mientras

//Ahora solo resta mostrar la solución
Mostramos costoTotal, la cantidad de conjuntos que tiene componentes y
el tamaño de tuberíasConstruidas

//Mostramos un elemento de cada conjunto de componentes
Por cada conjunto en componentes
        mostrar el primer elemento
Fin por

//Mostramos las tuberías construidas
Por cada tubería i en tuberíasConstruidas
        mostrar tuberíasConstruidas[i]
Fin por
```

Pseudocódigo de la estructura UnionFind:

```
Vamos a representar cada conjunto como un árbol.
Tengo dos arreglos de tamaño n, uno que representa el id al que pertenece el elemento i,
con i entre 0 y n. Y el otro representa el rank del elemento i.
El id lo que indica es quien es la raíz del árbol al que pertenece el elemento.
A la vez, tengo un entero llamado cantidadDeArboles que tiene tamaño n al inicializar.

//devuelve la raiz del arbol a la que pertenece un elemento
buscar(p)
  raiz = p
  mientras raiz != id[raiz]
          raiz = id[raiz]
  mientras p != raiz
          newp = id[p]
          id[p] = raiz
          p = newp

  return raiz

//une dos componentes
unir(x, y)
  i = buscar(x)
  j = buscar(y)

  //si son iguales ya estaban en la misma componente
  si i == j return

  //hace que la raiz del menor apunte a la del mayor
  si rank[i] < rank[j]
    id[i] = j
  sino
   id[j] = i
   si rank[i] == rank[j]
    rank[i]++;


  cantidadDeArboles--;


//devuelve true si dos elementos estan en el mismo arbol
conectados(x,y)
   return buscar(x) == buscar(y);


count()
        return cantidadDeArboles;
```

3.3. Complejidad propuesta

Como se ve en el pseudocódigo, la solución al problema se divide en tres partes, que son guardar los datos de entrada, encontrar la solución y mostrarla. El enunciado nos pide resolver todo en una complejidad estrictamente menor a $\mathcal{O}(n^3)$.

Al procesar la entrada, se guardan n, m y c en $\mathcal{O}(1)$, se crea un UnionFind con n conjuntos en $\mathcal{O}(n)$, y

luego se guardan las tuberías posibles con costo menor a c en una cola de prioridad. Las tuberías son a lo sumo m y encolar cada una tiene costo $\mathcal{O}(log(m))$. Por lo tanto, todo el proceso de guardar los datos de entrada tiene complejidad $\mathcal{O}(m * log(m))$, y como m está acotado superiormente por n^2, la complejidad sería $\mathcal{O}(n^2 * log(n))$.

Veamos ahora la complejidad de encontrar la menor solución. Esto se realiza en un ciclo definido por el tamaño de la cola de prioridad, que es a lo sumo m. Dentro del ciclo, se guarda el primer elemento de la cola, esto se realiza en $\mathcal{O}(1)$. Luego se revisa si ambas tuberías pertenecen al mismo conjunto del UnionFind, de no ser así, estas se unen.

Entonces, por cada tubería, tengo a lo sumo que llamar dos veces a la función buscar y una vez a la función unir del UnionFind.

Si la tubería es construida, además de unir en el UnionFind, se debe agregarla al vector que guarda las tuberías construidas, y esto se hace en $\mathcal{O}(1)$. También, se debe bajar el costo total de la construcción que también es $\mathcal{O}(1)$.

Como podemos ver en el pseudocódigo, nuestro UnionFind utiliza las dos heurísticas para UnionFind implementado sobre árboles. Éstas son union by rank y path compression.

La heurística path compression está implementada en la función buscar y lo que hace es que al buscar un elemento, les actualiza no solo a el elemento sino a todos sus padres la raíz a la raíz del árbol. Esto hace que en llamados posteriores no sea necesario recorrer todo el árbol.

La heurística union by rank para cada x, define rank[x] como alguna cota superior para la altura del subarbol con raíz en x y luego hace la unión por el tamaño de rank. Esto hace que la unión sea $\mathcal{O}(\log(n))$ porque cada conjunto tiene a lo sumo n elementos y el rank de un árbol sube solo al unirlo con otro de su mismo tamaño.

Combinando estas dos heurísticas, nos queda una complejidad total de encontrar la mejor solución de $\mathcal{O}(m * \alpha(n))$[4] donde α es el inverso de la función de Ackermann y está acotado por 5 para todo uso práctico. Ahora, como m está acotado superiormente por n^2, la complejidad sería $\mathcal{O}(n^2 * \alpha(n))$, y en particular, está acotado por $\mathcal{O}(n^2 * log(n))$.

Luego resta la parte de mostrar la solución, esto se hace en $\mathcal{O}(1)$ para mostrar el costo total, la cantidad de refinerías construidas y la cantidad de tuberías construidas. Se muestran en $\mathcal{O}(m) = \mathcal{O}(n^2)$ todas las tuberías que fueron construidas, ya que hay que recorrer el vector de tuberías. Y, por último, se realiza un ciclo que, por cada pozo, muestra el pozo si tiene una refinería en el mismo. Vamos a decir que un pozo tiene una refinería si es la raiz del árbol al que pertenece en el UnionFind. Entonces, si acotamos el buscar del UnionFind por $\mathcal{O}(\log(n))$ ya que el árbol está balanceado, tenemos una complejidad de $\mathcal{O}(m * \log(n))$, y como m está acotado superiormente por n^2, la complejidad sería $\mathcal{O}(n^2 * log(n))$.

Por lo tanto, sumando todas las étapas de la solución, nos quedaría:

$$\mathcal{O}(n^2 * log(n)) + \mathcal{O}(n^2 * \alpha(n)) + \mathcal{O}(n^2) + \mathcal{O}(n^2 * log(n))$$

O sea,

$$\mathcal{O}(2 * n^2 * log(n) + n^2 * \alpha(n) + (n^2) \in \mathcal{O}(n^2 * log(n))$$

Por lo tanto, nuestra complejidad teoríca final es

$$\mathcal{O}(n^2 * log(n))$$

Y ésta es estrictamente menor a $\mathcal{O}(n^3)$ como pedía el enunciado.

[4]La demostración se encuentra en Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. The MIT Press, 2001.

3.4. Implementación en C++

```
#include <vector>
#include <iostream>
#include <queue>
using namespace std;

class DisjointSet    {
    int *id, cantidadDeArboles, *rank;

public:

    DisjointSet(int N)   {
        cantidadDeArboles = N;

        id = new int[N];
        rank = new int[N];

        for(int i=0; i<N; i++)  {
            id[i] = i;
            rank[i] = 1;
        }
    }
    ~DisjointSet()   {
    delete [] id;
    delete [] rank;
    }

    // Devuelve el id de la componente a la que pertenece p
    int buscar(int p) {
        int raiz = p;
        while (raiz != id[raiz])
            raiz = id[raiz];
        while (p != raiz) {
            int newp = id[p];
            id[p] = raiz;
            p = newp;
        }
        return raiz;
    }
    // mergea dos sets
    void unir(int x, int y)    {
        int i = buscar(x);
        int j = buscar(y);
        if (i == j) return;

        //hace que la raiz del menor apunte a la del mayor
        if(rank[i] < rank[j]){
            id[i] = j;
        }else{
            id[j] = i;
            if(rank[i] == rank[j]){
                rank[i]++;
            }
```

```
        }
        cantidadDeArboles--;
    }

    //devuelve true si dos pozos estan en el mismo arbol
    bool conectados(int x, int y)    {
        return buscar(x) == buscar(y);
    }

    // Devuelve la cantidad de arboles
    int count() {
        return cantidadDeArboles;
    }

};



struct Tuberia {
    Tuberia(): desde(0), hasta(0), costo(0){}
    Tuberia(const int d,const int h,const int c) : desde(d), hasta(h),
       costo(c) {};

    // desde que pozo va la tuberia
    int desde;

    //hasta que pozo va la tuberia
    int hasta;

    //costo de la construccion
    int costo;
};

struct compararPorCostos
{
  bool operator()(const Tuberia& izq, const Tuberia& der) const
  {
    return izq.costo > der.costo;
  }
};

typedef vector<Tuberia> Vec;

// Prototipado de funciones


// Implementacion. Contiene el cargado de input mas la resolucion del
   ejercicio.
int main() {
    // cantidad de pozos
    int n;
    cin >> n;

    // cantidad de pares de pozos en los cuales se puede construir una
```

```
    tuberia
int m;
cin >> m;

// costo de construir una refineria
int c;
cin >> c;

// cargo las potenciales tuberias en una cola de prioridad y guardo
    en un vector si puedo
//acceder por tuberia o necesito refineria si o si.
std::priority_queue<Tuberia, std::vector<Tuberia>, compararPorCostos
    > tuberias;

//defino el costo total como el costo de poner una refineria por
    pozo
int costoTotal = n*c;

//O(m*log(m))
for(int i = 0; i < m; i++) {
    int desde, hasta, costo;
    cin >> desde;
    cin >> hasta;
    cin >> costo;

    //solo agrego las tuberias que tengan costo menor a la refineria
        ya que
    //solo es conveniente por costos hacerlo si es asi

    if(costo < c){
        tuberias.push(Tuberia(desde,hasta,costo));
    }
}

DisjointSet componentes(n);

//en este vector se guardaran las tuberias que se vayan construyendo
    , tiene como maximo
//el tamano de tuberias que agregue a tuberias, es decir, la
    cantidad de tuberias con costo menor a c
vector<Tuberia> tuberiasConstruidas;
tuberiasConstruidas.reserve(tuberias.size());

cout << endl;
while(!tuberias.empty()){
    Tuberia t = tuberias.top();
    tuberias.pop();

    //busco con -1 para que encuentre el indice correcto en el UF
    if(!componentes.conectados(t.desde - 1,t.hasta - 1)){

        //uno las componentes en el disjoint set
        componentes.unir(t.desde - 1,t.hasta - 1);
```

```
            //agrego la tuberia a tuberias construidas
            tuberiasConstruidas.push_back(t);

            //bajo el costo total
            costoTotal = costoTotal - c + t.costo;
        }
    }

    cout << costoTotal << "␣" <<  componentes.count() << "␣" <<
       tuberiasConstruidas.size() << endl;

    //imprimo los pozos que tienen refineria
    //defino que un pozo tiene refineria si es la raiz de su componente
    for (int i = 0; i < n; i++){
        if(componentes.buscar(i) == i){
            cout << i + 1 << "␣";
        }
    }

    //imprimo las tuberias que se construyeron
    cout << endl;
    for(int i = 0; i < tuberiasConstruidas.size();i++){
        cout << tuberiasConstruidas[i].desde << "␣" <<
           tuberiasConstruidas[i].hasta << endl;
    }

}
```

3.5. Experimentación computacional

La función que utilizamos para llevar a cabo las mediciones fue `std::clock`[5]. La unidad temporal que utilizamos para este ejercicio fue milisegundos. La complejidad teórica calculada es de $\mathcal{O}(n^2 * \log(n))$

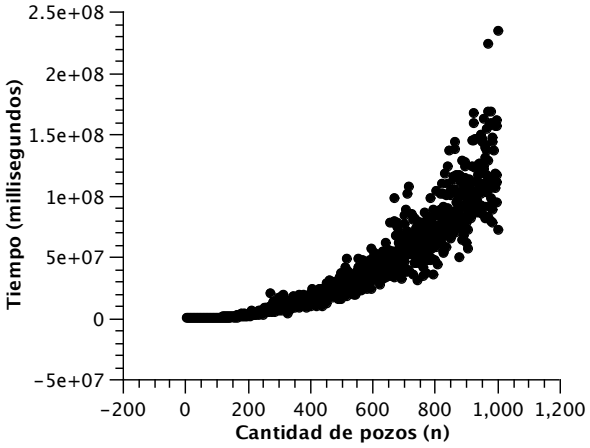
3.5.1. Experimentación con instancias aleatorias

Para generar las instancias aleatorias utilizamos la función `std::rand`[6] con determinados intervalos de valores para la variables, para obtener instancias coherentes. El detalle de intervalos es el siguiente:

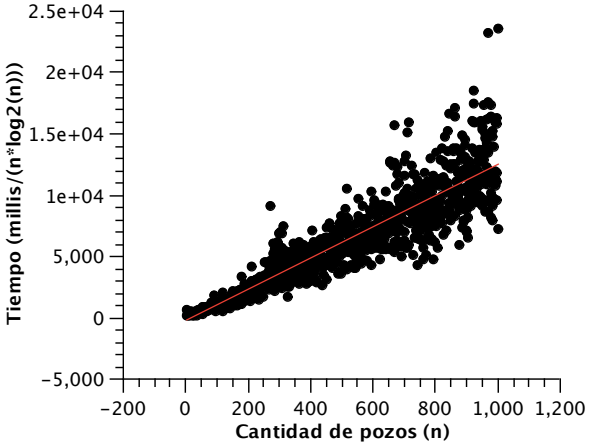
- Cantidad de pozos (n): $2 \leq n \leq 1000$
- Cantidad de tuberías (m): $1 \leq m \leq \frac{n(n-1)}{2}$
- Costo de refinería (c): $1 \leq c \leq 1000$

No se generaron muestras para n = 1 ya que no hay tuberías posibles. Generamos 20 instancias aleatorias para cada n, variando aleatoriamente el m y el c en cada muestra y luego fueron promediadas. Su medición temporal, arroja el siguiente resultado:

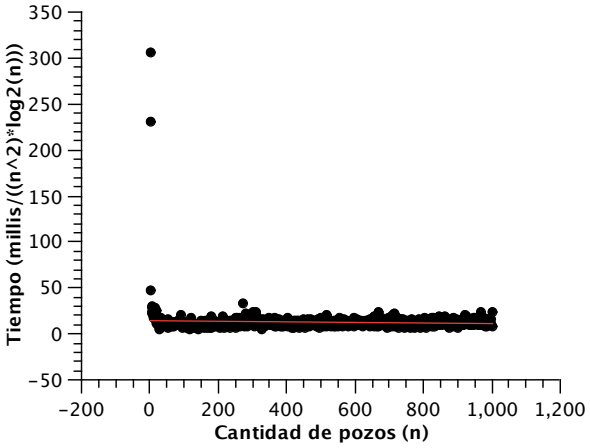
[5]Referencia `http://en.cppreference.com/w/cpp/chrono/c/clock`
[6]Referencia `http://en.cppreference.com/w/cpp/numeric/random/rand`



Tiempos sin procesar, en milisegundos



Dividiendo a los tiempos por $n \log(n)$



Dividiendo a los tiempos por $n^2 \log(n)$

A continuación, adjuntamos una tabla con los últimos 20 valores obtenidos en las instancias aleatorias, teniendo en cuenta que los casos fueron previamente ordenados según el tamaño (n):

n	Tiempo(milisegundos)	Tiempo(mili/($n^2 \log(n)$))	Tiempo(mili/($n \log(n)$))
980	106,884,913.7	11.20017462183745	10,976.17112940071
981	148,013,000.5	15.47597708270933	15,181.93351813786
982	143,649,318.15	14.98692707630785	14,717.16238893431
983	88,530,761.25	9.21626618012942	9,059.589655067221
984	77,926,122.55	8.09462319280639	7,965.109221721488
985	96,363,820.55	9.98806386893941	9,838.242910905317
986	137,058,449.3	14.17515494504867	13,976.70277581799
987	109,314,001.3	11.28115224428363	11,134.49726510795
988	116,393,678.25	11.98570806428229	11,841.8795675109
989	95,277,182.2	9.789957451616216	9,682.267919648439
990	106,078,816.3	10.87624855027109	10,767.48606476838
991	156,976,954.55	16.0600124239817	15,915.47231216586
992	112,528,406.3	11.48768775240836	11,395.7862503891
993	117,798,450.6	11.99972974500853	11,915.73163679347
994	109,700,234.25	11.1506922114351	11,083.78805816649
995	156,578,939.15	15.88147949436476	15,802.07209689294
996	117,020,018.95	11.84355454694666	11,796.18032875887
997	94,736,198.95	9.567601826537732	9,538.89902105812
998	161,853,906.15	16.31084667818704	16,278.22498483067
999	110,925,527.35	11.15454731027924	11,143.39276296896
1,000	72,523,403.45	7.27723994203022	7,277.239942030219

Como podemos ver de los gráficos y la tabla suministrada,al dividir los tiempos por $n * \log(n)$, tienden a algo lineal, mientras que al dividir los tiempos por $n^2 * \log(n)$, tienden a un número constante. Entonces nuestro algoritmo tendría complejidad $\mathcal{O}(c * n^2 * \log(n))$, donde c es la constante a la cual converge el gráfico. Por lo tanto concluimos que los gráficos se condicen con nuestra predicción de complejidad.

3.5.2. Experimentación con instancias particulares

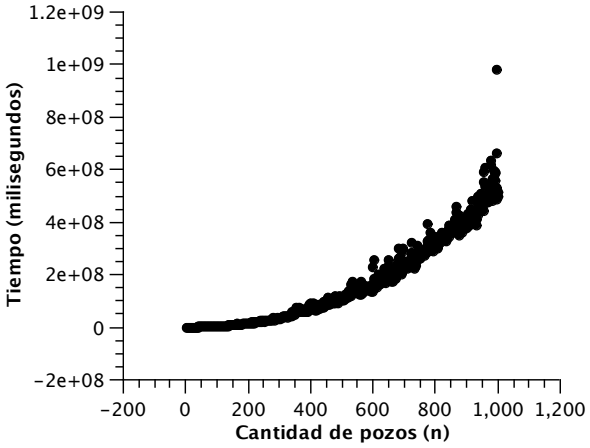
Para este problema se nos ocurre probar con instancias de mejor y peor caso.

Una instancia de peor caso sería una donde dado n, las tuberías posibles m sean $\frac{n(n-1)}{2}$ y que todas tengan costo menor a c, es decir, que dado un pozo, sea posible construir una tubería entre ese pozo y cualquier otro con costo menor al de una refinaría.

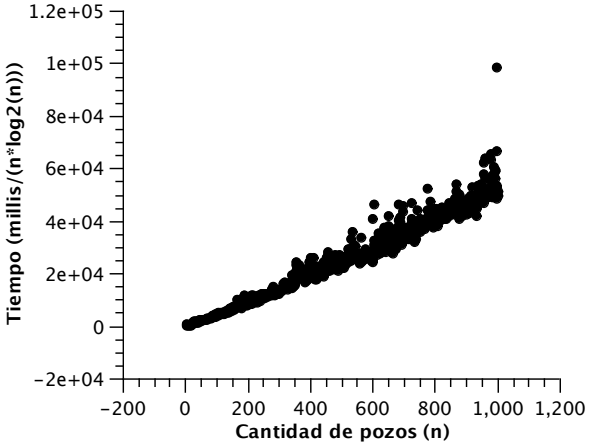
El detalle de intervalos es el siguiente:

- Cantidad de pozos (n): $2 \leq n \leq 1000$
- Cantidad de tuberías (m): $\frac{n(n-1)}{2}$
- Costo de refinería (c): $1 \leq c \leq 1000$

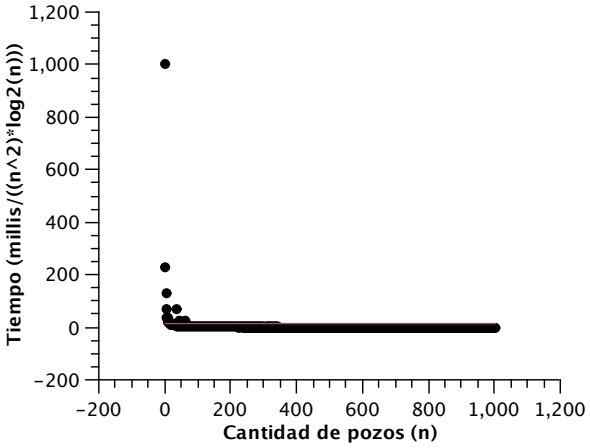
No se generaron muestras para n = 1 ya que no hay tuberías posibles. Generamos 20 instancias aleatorias para cada n, variando el c en cada muestra pero fijando para toda tubería un costo de $c - 1$ en cada muestra y luego fueron promediadas. Su medición temporal, arroja el siguiente resultado:



(a) Tiempos sin procesar, en milisegundos



(b) Dividiendo a los tiempos por $n \log(n)$



(a) Dividiendo a los tiempos por $n^2 \log(n)$

A continuación, adjuntamos una tabla con los últimos 20 valores obtenidos en las instancias aleatorias, teniendo en cuenta que los casos fueron previamente ordenados según el tamaño (n):

n	Tiempo(milisegundos)	Tiempo(mili/($n^2 \log(n)$))	Tiempo(mili/($n \log(n)$))
980	493,460,879.75	50,674.23338154993	0.08452622007433862
981	482,018,120.4	49,441.38037692552	0.09101871660609798
982	510,165,009.05	52,267.43419346325	0.08472145916999867
983	516,396,402.7	52,844.22546197011	0.08947293513434787
984	561,265,941.75	57,369.01031618338	0.07824592057934007
985	490,971,048.75	50,125.58045390238	0.1026006503166199
986	525,554,033.5	53,594.01449803058	0.09532040618344501
987	570,880,765.55	58,148.72977958777	0.1056907440396239
988	596,367,785.8	60,674.39059893046	0.08665430150997189
989	566,389,457.35	57,557.68953594214	0.09656804009648187
990	508,075,400.3	51,571.981884774	0.09234514821696686
991	583,108,179.7	59,119.77408159488	0.1045670681684471
992	587,978,851.1	59,544.79875082848	0.0944663358419995
993	557,400,980.3	56,383.08875465296	0.1324639066988179
994	590,795,854.55	59,692.27032416804	0.07456885483672317
995	979,012,067	98,802.68285405725	0.1032966094149012
996	662,148,109.4	66,747.71182669644	0.09686879345266487
997	532,542,101.5	53,621.16473927397	0.1027177598861123
998	525,905,149.05	52,892.15775233119	0.09732184303195211
999	486,305,604.75	48,853.44686677655	0.1106853151800258
1,000	497,043,395.25	49,874.99037230599	0.08448103791447331

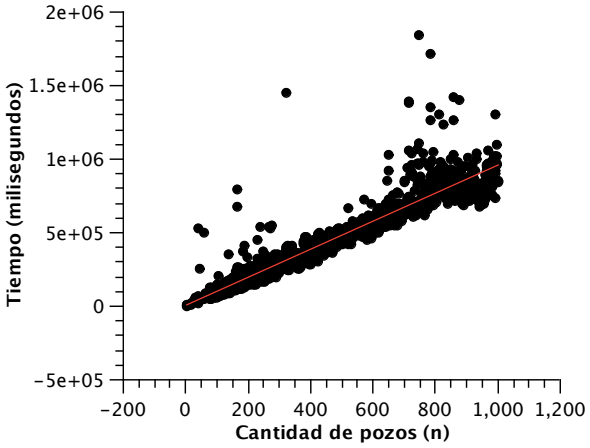
Como podemos ver de los gráficos y la tabla suministrada, al dividir los tiempos por $n*\log(n)$, tienden a algo lineal, mientras que al dividir los tiempos por $n^2*\log(n)$, tienden a un número constante. Entonces nuestro algoritmo tendría complejidad $\mathcal{O}(c*n^2*\log(n))$, donde c es la constante a la cual converge el gráfico y esto se cumple incluso en el peor caso. Por lo tanto concluimos que los gráficos se condicen con nuestra predicción de complejidad.

Por otro lado, una instancia de mejor caso sería una donde dado n, las tuberías posibles m sean una cantidad aleatoria y que todas tengan costo mayor a c.

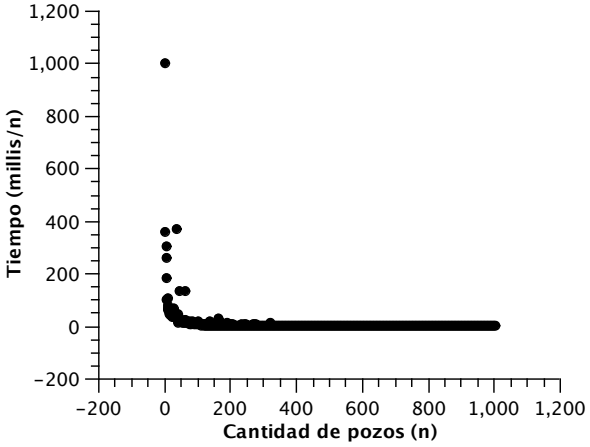
El detalle de intervalos es el siguiente:

- Cantidad de pozos (n): $2 \leq n \leq 1000$
- Cantidad de tuberías (m): $1 \leq m \leq \frac{n(n-1)}{2}$
- Costo de refinería (c): $2 \leq c \leq 1000$

No se generaron muestras para n = 1 ya que no hay tuberías posibles. Generamos 20 instancias aleatorias para cada n, variando el c en cada muestra pero fijando para toda tubería un costo mayor a $c-1$ en cada muestra y luego fueron promediadas. Su medición temporal, arroja el siguiente resultado:



(a) Tiempos sin procesar, en milisegundos



(b) Dividiendo a los tiempos por n

A continuación, adjuntamos una tabla con los últimos 20 valores obtenidos en las instancias aleatorias,

teniendo en cuenta que los casos fueron previamente ordenados según el tamaño (n):

n	Tiempo(milisegundos)	Tiempo(mili/(n))
980	806,646.15	823.1083163265306
981	870,507.45	887.3674311926605
982	812,053.05	826.9379327902241
983	859,470.3	874.3339776195321
984	753,265.6	765.5138211382114
985	989,880.6	1,004.954923857868
986	921,645.45	934.7316937119675
987	1,024,139.9	1,037.629078014184
988	841,503.3	851.723987854251
989	939,813.15	950.2660768452984
990	900,665.7	909.7633333333333
991	1,022,080.15	1,031.362411705348
992	925,351.25	932.8137600806451
993	1,300,366.2	1,309.532930513595
994	733,606.55	738.0347585513078
995	1,018,423.6	1,023.541306532663
996	957,110.3	960.9541164658635
997	1,017,087.7	1,020.1481444333
998	965,732.85	967.6681863727455
999	1,100,701.5	1,101.803303303303
1,000	841,919.8	841.9198
1,001	849,901	849.0519480519481

Como podemos ver de los gráficos y la tabla suministrada, al dividir los tiempos por n, tienden a un número constante. Entonces nuestro algoritmo tendría complejidad $\mathcal{O}(c * n)$, donde c es la constante a la cual converge el gráfico. Esto sucede porque como todas las tuberías tienen costo mayor a c, ninguna es añadida a la cola de prioridad, y por ende no es necesario solucionar el problema ya que la solución de menor costo es poner una refinería en cada pozo. El algoritmo es $\mathcal{O}(n)$ en mejor caso porque es lo que cuesta el ciclo que recorre el UnionFind para ver donde va una refinería, que en este caso es $\mathcal{O}(1)$ porque todas se tienen como raíz a si mismas.