Algoritmos y estructuras de datos III

Trabajo Práctico Nº2

De Sousa Bispo Mariano	389/08	marian_sabianaa@hotmail.com
Grosso Daniel	694/08	dgrosso@gmail.com
Livorno Carla	424/08	carlalivorno@hotmail.com
Raffo Diego	423/08	enanodr@hotmail.com

Mayo 2010

Índice

Introducción

Este trabajo tiene como objetivo la aplicación de diferentes técnicas algorítmicas para la resolución de tres problemas particulares, el cálculo de complejidad teórica en el peor caso de cada algoritmo implementado, y la posterior verificación empírica.

El lenguaje utilizado para implementar los algoritmos de todos los problemas fue C/C++

1. Problema 1

Sea $s = (s_1, s_2, .., s_n)$ una secuencia de números enteros. Determinar la mínima cantidad de elementos de s tales que al ser eliminados de la secuencia, el resto de los elementos forman una secuencia unimodal.

1.1. Explicación

La resolución del problema consiste en considerar cada uno de los elementos de la secuencia dada como posible máximo de una subsecuencia (al que nos referiremos como "pico"), tal que sea unimodal. Cada subsecuencia unimodal tiene longitud máxima, es decir, contiene la subsecuencia creciente hasta el pico y la subsecuencia decreciente desde el pico, ambas con la mayor cantidad de elementos posibles. Al no necesitar explicitar los elementos de la secuencia, la única información que aporta a la solución del problema es la longitud de cada subsecuencia. De esta manera, para determinar la mínima cantidad de elementos que se deben eliminar para transformar la secuencia dada en unimodal, basta con conocer la diferencia entre la longitud de la secuencia original y el máximo de las longitudes de cada subsecuencia unimodal.

1.2. Detalles de la implementación

Para determinar la máxima longitud de la subsecuencia creciente y decreciente hasta cada elemento, utilizamos la técnica de programación dinámica.

Sea la secuencia dada $S = [s_1, s_2, ..., s_n]$ y $C = [c_1, c_2, ..., c_n]$ con $c_i = \max_{0<j<i}\{c_j / s_j < s_i\} + 1$ $(\forall\ i \in \mathbb{N}, 0 < i \leq n)$, es decir, para cada i tenemos en c_i la máxima longitud de la subsecuencia creciente que incluye a s_i. Análogamente, se define D como la secuencia que contiene las máximas longitudes de las subsecuencias decrecientes de S.

El algoritmo que calcula la solución implementa tanto C como D. Para obtener la máxima longitud de la subsecuencia creciente hasta el índice i, itera por todos los índices $j < i$ en C, buscando el máximo valor entre los c_j tales que s_i sea mayor que s_j. Esto asegura que en la posición c_i está la máxima longitud de la subsecuencia creciente que incluye a s_i ya que, si existiese otra subsecuencia de mayor longitud a la que se pueda agregar s_i, se podría agregar el s_i a esa secuencia y así obtener una con más cantidad de elementos, siendo el valor de c_i la longitud de dicha secuencia más 1. Para calcular D, invertimos S y aplicamos el mismo procedimiento que para C, quedando en d_i la máxima longitud de la subsecuencia decreciente que incluye a s_{n-i}.

Luego de calcular C y D el algoritmo determina la mínima cantidad de elementos a ser eliminados de la secuencia tales que el resto de los elementos forman una secuencia unimodal, de la siguiente forma:

```
secuencia_unimodal(secuencia, C, D)
    max_long ← 0
    for i = 1  to n
        long_secuencia_unimodal ← C[i] + D[n − i] − 1
        if long_secuencia_unimodal > max_long
            max_long ← long_secuencia_unimodal
    return longitud(secuencia)-max_long
```

1.3. Análisis de complejidad

Como hemos visto anteriormente, nuestro algoritmo utiliza la técnica de programación dinámica. Esta consta de reutilizar información previamente calculada para llegar al resultado final. En el análisis de la complejidad veremos como la aplicación de la técnica previamente mencionada, ha permitido conseguir un algoritmo polinomial.

El algoritmo *long_max_creciente* es el que utiliza la técnica de programación dinámica, calculando para cada elemento i de la secuencia original la máxima longitud de la subsecuencia creciente y decreciente que lo incluye, iterando por todos los índices $j < i$ ($\forall i\ 0 \leq i < n$), sabiendo que para cada j ya esta calculada la longitud de la subsecuencia creciente más larga que lo incluye.

Sea n la longitud de la secuencia dada, *max_long_creciente* la secuencia que guarda en cada posición la máxima longitud de la subsecuencia creciente.

```
long_max_creciente(secuencia, max_long_creciente)
    max_long_creciente[0] ← 0                                   O(1)
    for i = 1 to n
        max_long ← 0                                            O(1)
        for j = i − 1 to 0                                      O(i)
            if secuencia[j] < secuencia[i] and                  O(1)
            max_long_creciente[j] > max_long)
                max_long ← max_long_creciente[j]                O(1)
        max_long_creciente[i] ← max_long + 1                    O(1)
```

Podemos ver dentro de cada ciclo (for) que todas las asignaciones tienen costo constante, así como también la guarda del if. De esta manera podemos concluir que por cada iteración del for anidado tenemos en el peor caso el costo de la guarda del if, más el costo de la asignación dentro del mismo, con costo constante.

El ciclo anidado iterará para cada i, $i-1$ veces, siendo los valores posibles de i desde 1 hasta n.

La complejidad de dicho algoritmo viene dada por: $\sum_{i=0}^{n-1} i = \frac{n*(n-1)}{2}$

Esta sumatoria, se coindice entonces con el costo de la función *long_max_creciente*. La cantidad de operaciones que realiza este algoritmo es $O(n^2)$.

El algoritmo encargardo de devolver el resultado del problema es *secuencia_unimodal*, el cual utiliza el algoritmo descripto anteriormente y realizar algunas otras operaciones que se detallarán a continuación para concluir con el análisis de complejidad.

```
secuencia_unimodal(secuencia, n)
    C[n]
    D[n]
    R ← reverso(secuencia)                                  O(n)
    long_max_creciente(secuencia, C)                        O(n^2)
    long_max_creciente(R, D)                                O(n^2)
    max_long ← 0                                            O(1)
    for i = 1  to n                                         O(n)
        long_secuencia_unimodal ← C[i] + D[n − i] − 1       O(1)
        if long_secuencia_unimodal > max_long               O(1)
            max_long ← long_secuencia_unimodal              O(1)
    return longitud(secuencia)-max_long                     O(1)
```

La función reverso que toma una secuencia y devuelve otra con los elementos en orden inverso tiene costo lineal, en función de la cantidad de elementos, ya que itera una vez por la secuencia original ($desde\ i = 0\ hasta\ n - 1$), guardando cada valor en la posición $n-1-i$ de la secuencia resultante. Cabe destacar que el costo de la asignación es constante.

El ciclo perteneciente a *secuencia_unimodal* (for), itera n veces, siendo en el peor caso el costo de cada iteración acotado por una constante. La cantidad máxima de operaciones se da en las iteraciones que entra al caso condicional. El costo del for es $n * c$ (con c constante), es decir, O(n).

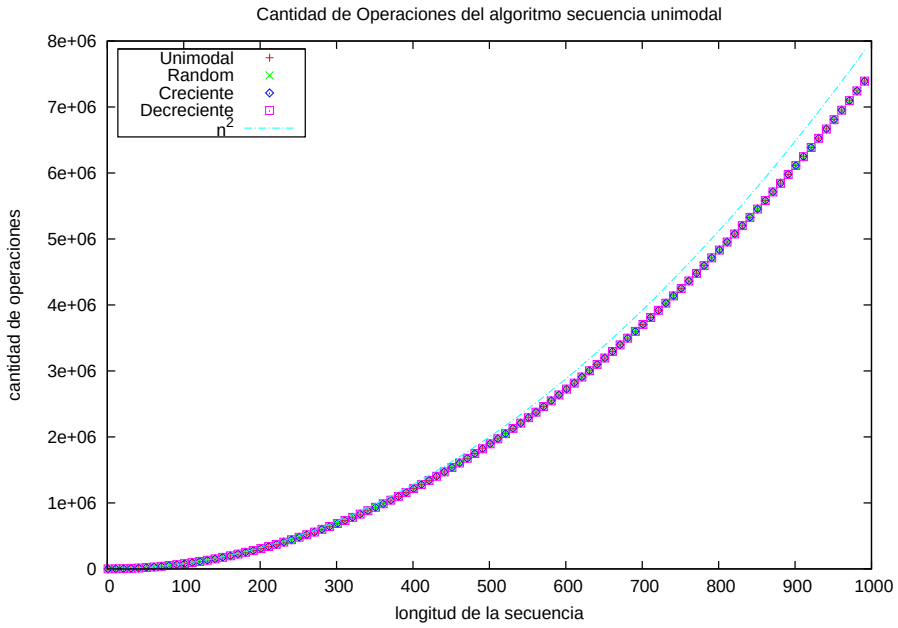
Por lo tanto, el costo del algoritmo *secuencia_unimodal* es: O($2*n+2*n^2$) que por definición es O($max(2 * n, 2 * n^2)$) = O($2 * n^2$) = O(n^2).

1.4. Pruebas y Resultados

Para probar correctitud tenemos un generador de secuencias unimodales, secuencias crecientes y decrecientes que nos devuelve en un archivo $test\_*.in$, donde $*$ se refiere a *unimodal*, *creciente o decreciente* según corresponda. De esta forma, con una comparación de archivos (comando `make diffs` en la carpeta Ej1) podemos saber para cada instancia si el resultado obtenido por nuestro algoritmo es correcto.

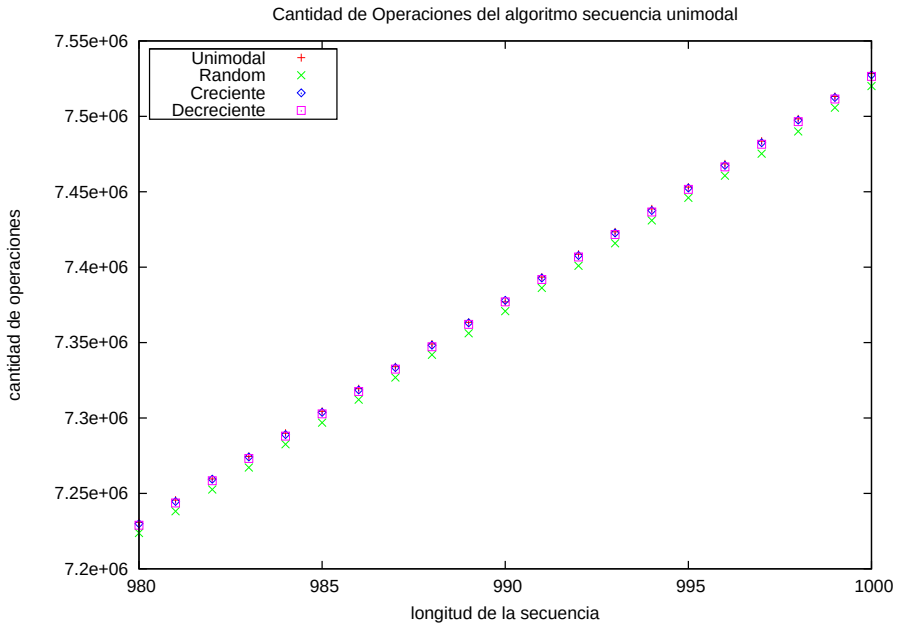
Para llevar a cabo las pruebas de este algoritmo en cuanto a operaciones realizadas por instancia, generamos secuencias unimodales, crecientes, decrecientes y random con el objeto de constatar si los gráficos (a continuación expuestos), se condicen con la complejidad teórica analizada.

En el siguiente gráfico, se ven sólo las instancias con n múltiplo de 10 para poder apreciar cada instancia para entrada unimodal, creciente, decreciente y random (si graficáramos todos los n se vería una linea densa y no podríamos apreciar el comportamiento del algoritmo).



Podemos observar que para un n dado, todos los casos vistos, coinciden en la cantidad de operaciones. Además la curva n^2 parece ser una buena cota para el algoritmo. Se aprecia también que a medida que n crece, la diferencia entre n^2 y la cantidad de operaciones para nuestro algoritmo es cada vez mayor. Podemos decir entonces, que n^2 es una buena cota, pero que a medida que n crece, el valor de la constante c (coeficiente cuadrático) que fue despreciado, así como también su término lineal y el constante (en el análisis de complejidad) contribuye a que la diferencia entre las dos curvas siga creciendo.

El siguiente gráfico, muestra la cantidad de operaciones para n entre 980 y 1000. Queremos ver si existe diferencia entre los distintos tipos de test, ya que en el gráfico anterior no pudimos ver estos resultados por la escala en la que trabajamos, y no podemos afirmar entonces que no existe tal diferencia.



Corroboramos que a pesar de que es mínima, existe diferencia entre los distintos tipos de casos. Sabemos que la cantidad de operaciones para cada n del gráfico es del orden de millones. La diferencia entre los diferentes tipos de secuencias para cada n es de entre 1000 y 10000. A pesar de ser una gran diferencia de operaciones, en relación a la cantidad total, es *despreciable*.

1.5. Concluciones

Como antes expusimos, para resolver el ejercicio, se utilizó la técnica de Programacion Dinámica, sin la cual no podríamos haber cumplido la complejidad impuesta, utilizando su gran ventaja de poder guardar los resultados anteriores en estructuras poco costosas.

Luego de analizar el algoritmo de forma teórica y apreciar el comportamiento de este en diversas instancias en los gráficos, podemos concluir que el algoritmo encontrado para resolver el problema de la secuencia unimodal es adecuado, ya que pudimos mostrar que trabaja en complejidad polinomial, del orden de n^2, lo que significa que está 'bien' resuelto computacionalmente.

2. Problema 2

El gobierno planea construir una ciudad. Lo único que falta es asignarle el sentido de circulación a las calles (puede asignarse sólo un sentido a cada una). Toda calle está conectada a dos esquinas, y se quieren poder determinar si se puede llegar de cualquier esquina a cualquier otra.

2.1. Explicación

Para obtener el resultado para el problema, se toma una esquina y se intenta volver a la misma recorriendo otras esquinas, transitando las calles de la posible ciudad. Si no podemos volver a la esquina de la que partimos, podemos concluir que el resultado del problema será: *"No existe forma de asignar el sentido de circulación a las calles para poder llegar de cualquier esquina a cualquier otra"* . Si podemos volver a la esquina desde la que partimos, podemos afirmar que todas las esquinas que forman parte del circuito, son esquinas accesibles de cualquier otra esquina de ese mismo circuito. Resta ver entonces, que para el resto de las esquinas que no forman parte de ese circuito, se puede partir de alguna de las pertenecientes al circuito y volver, transitando por alguna o todas de las esquinas que no pertenecían al circuito.

Este proceso se debe realizar para todas las esquinas, agrandando en cada paso el conjunto de esquinas que forman parte del circuito (siempre y cuando no podamos afirmar que **no** existe solución previamente). Si esto ocurre, la solución al problema será: *"Existe forma de asignar el sentido de circulación a las calles para poder llegar de cualquier esquina a cualquier otra".*

2.2. Detalles de la implementación

Modelamos este problema mediante grafos donde los vértices corresponden a esquinas y las aristas a las calles que conectan dos esquinas.

Generamos la matriz de adyacencia del grafo, de $n \times n$ donde n es la cantidad de esquinas. Cada posición (i, j) de la matriz contiene un *uno* si la esquina i está conectada con la esquina j por medio de una calle, y *cero* en caso contrario. Nos referiremos a la matriz como *conexiones*.

Para la resolución del problema recorrimos el grafo con una modificación del algoritmo *Depth First Search*. Nuestro algoritmo, en primera instancia, busca el primer ciclo que pueda encontrar. Si no llega a la solución y recorre todo los nodos del grafo, puede afirmar que el resultado es falso, ya

que el grafo **no** es fuertemente conexo. En caso contrario, marca todos los nodos que componen el circuito simple como parte de los nodos que ya se encuentran conectados y entra a un ciclo (*while*). Este último, se encarga de buscar alguna arista que lleve a algún nodo todavía no perteneciente a la solución parcial. Si la encuentra, aplica dfs desde ese vértice hasta que vuelva al ciclo, si no la encuentra y todavía quedan nodos por recorrer, no hay solución (devuelve $falso$). El dfs (en caso de que exista una arista libre) busca volver al ciclo original, y si lo hace, agrega todos los nodos por los que pasó como parte del ciclo. Vuelve a empezar hasta que haya pasado por todos los nodos (*devuelve fuertemente conexo*) o algún nodo no puede volver al ciclo (*devuelve no*). Cabe destacar que por cada vez que el dfs encuentre un camino que sale y vuelve del ciclo, el nuevo ciclo crece (se agregan todos los nodos del camino resultante).

El siguiente pseudocódigo detalla lo explicado anteriormente.

```
ciudad(conexiones)
    ciclo[0...n] ← false
    encontre_ciclo ← dfs_primer_ciclo(conexiones, ciclo)
    for j to n
        termine ← termine and ciclo[j]
    while !termine and encontre_ciclo
        nodo_busqueda, nodo_salida ← adyacente_externo(conexiones, ciclo)
        conexiones[nodo_busqueda][nodo_salida] ← 0
        encontre_ciclo ← dfs_ciclo(conexiones, nodo_busqueda, ciclo)
        conexiones[nodo_busqueda][nodo_salida] ← 1
        termine ← true
        for i to n
            termine ← termine and ciclo[i]
    return termine
```

- Tenemos la variable de tipo *bool* **termine**, que nos indica si todos los vértices del grafo ya son considerados parte del ciclo. Esta variable se

setea, cada vez que encontramos un ciclo (ya sea el primero, o cuando se agregan vértices al principal).

- **dfs_primer_ciclo:** La modificación que le hicimos al dfs en este caso consiste en numerar los vértices. Esto nos permite determinar los vértices pertenecientes al circuito una vez que lo detectamos. Para encontrar el primer circuito, cada vez que sacamos un vértice i de la pila le asignamos el número de su predecesor más *uno* y verificamos que cada vértice tenga algún adyacente todavía no numerado (podría ser que ya esté visitado), ya que si esto no ocurre sabemos que no existe tal ciclo porque lo que tenemos hasta ahí es un camino y no hay forma de volver (tiene conexión sólo con su predecesor). Si en algún momento un vértice adyacente a i ya fue visitado y su número es menor al de i menos *uno* (es decir, el adyacente ya visitado no es su predecesor) decimos que hay un ciclo y este está formado por los vértices que tienen menor numeración que i los cuales son marcados en el arreglo *ciclo*. Además, esta función devuelve un *bool* que nos dice si encontró un circuito (*encontre_ciclo*). Si *encontre_ciclo* es falso significa que no existen circuitos en el grafo por lo que ya podemos afirmar que no es fuertemente conexo.

- **adyacente_externo:** Esta función se encarga de buscar un vértice que todavía no pertenezca al circuito tal que sea adyacente a uno perteneciente. Seteando la variable *nodo_busqueda* con el vértice no perteneciente al circuito principal y *nodo_salida* con el vértice que si lo es.

- **dfs_ciclo:** En este caso también numeramos los vértices para determinar cuáles son los pertenecientes al circuito, si es que se puede volver al circuito principal sin usar la arista que conecta el vértice *nodo_busqueda* con *nodo_salida*. Garantizamos que no use esa arista para volver, marcándola como 0 en la matriz de adyancencia previo ingreso a dfs_ciclo. El valor original de la arista en la matriz de adyacencia es restaurado una vez que la función termina.

  La función apila el *nodo_busqueda*. Apila sus adyacentes (si es que tiene alguno distinto al *nodo_salida*) y continúa con el algoritmo dfs. Termina si encuentra un vértice adyacente que no esta visitado por el dfs_ciclo y forma parte del circuito principal, o si la pila se vacía y todavía no logro volver al circuito.

2.3. Análisis de complejidad

A continuación se enumera los costos de las partes del algoritmo ciudad:

- Al inicio el algoritmo tiene un ciclo for para inicializar el arreglo *ciclo* (de tamaño n) el cual tiene un costo de n.
- La función dfs_primer_ciclo tiene un costo de n^2 ya que a lo sumo visita una vez cada vértice recorriendo para cada uno de estos la columna correspondiente en la matriz de adyacencia.
- Luego de setear la variable *encontre_ciclo* con el resultado de la función dfs_primer_ciclo el algoritmo verifica si todos los vértices ya pertenecen a un circuito, esto lo hace con un ciclo for el cual recorre todo el arreglo *ciclo* de tamaño n. Por lo que el costo de este ciclo es n.
- Dentro el algoritmo *ciudad* tenemos un ciclo *while* que itera a lo sumo $n - 3$ veces (encuentra el ciclo principal de tres vértices y dentro del ciclo el dfs_ciclo agrega un vértice por iteración). Dentro de este ciclo, hay dos llamadas a funciones (*adyacente_externo* y dfs_ciclo), un ciclo for de *uno* hasta n (con costo lineal),y una cantidad constante de asignaciones, comparaciones e indexación con costo constante. Entonces el costo del *while* resulta ser el máximo entre los costos mencionados y el costo de las funciones por n (ya que es el orden de la cantidad de iteraciones del while).

    - La función *adyacente_externo* tiene un costo del orden de n^2 porque recorre todos los vértices y para cada uno de estos que pertenece al circuito busca entre todos los vértices alguno que sea adyacente a él y no pertenezca al circuito (no termina apenas encuentre uno, sino que los recorre todos y se queda con el último que cumple la condición).
    - Por otro lado, la función dfs_ciclo posee la misma complejidad que dfs_primer_ciclo. Dado un vértice, a lo sumo visitará todo los demás menos los que ya pertenecen al circuito principal (a lo sumo serán $n - 3$ vértices). Para cada uno de estos, deberá obtener sus adyacentes, los que pueden ser n.

Podemos decir entonces que el ciclo del algoritmo *ciudad* tiene costo del orden de n^3.

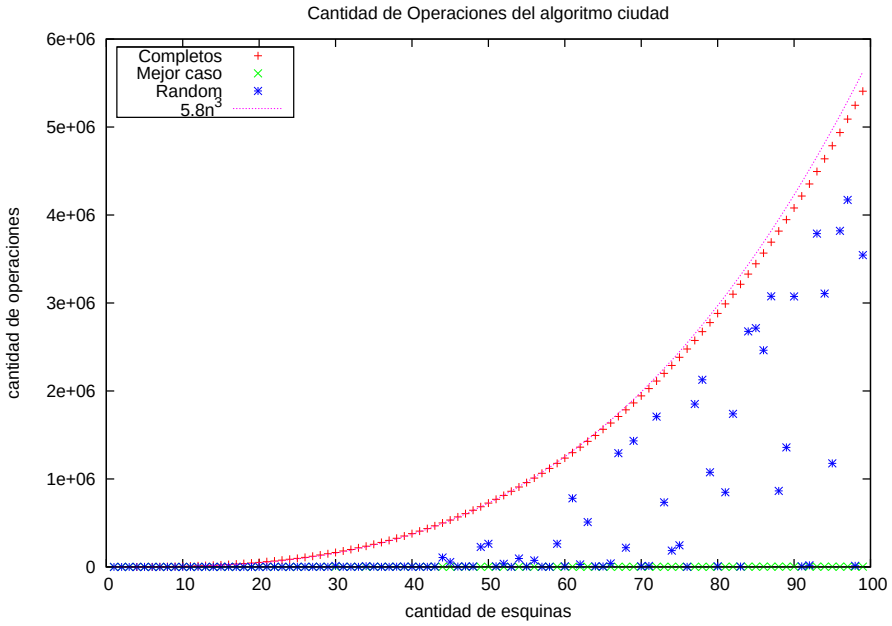
La complejidad del algoritmo *ciudad* viene dada por el máximo entre el costo del ciclo (*while*), dos ciclos *for* (iteran de *uno* hasta n con operaciones de costo constante dentro) y la llamada a al función *dfs_primer_ciclo*. Por lo tanto podemos afirmar que la complejidad asintóntica es O(n^3).

2.4. Pruebas y Resultados

Para probar correctitud contamos con un generador de instancias con grafos fuertemente conexos y otras no fuertemente conexos. Estas entradas se guardan en los archivos `test_fc.in` y `test_nofc.in`. De esta forma, con una comparación de archivos (comando `make diffs` en la carpeta Ej2) podemos saber para cada instancia si el resultado obtenido por nuestro algoritmo es correcto.

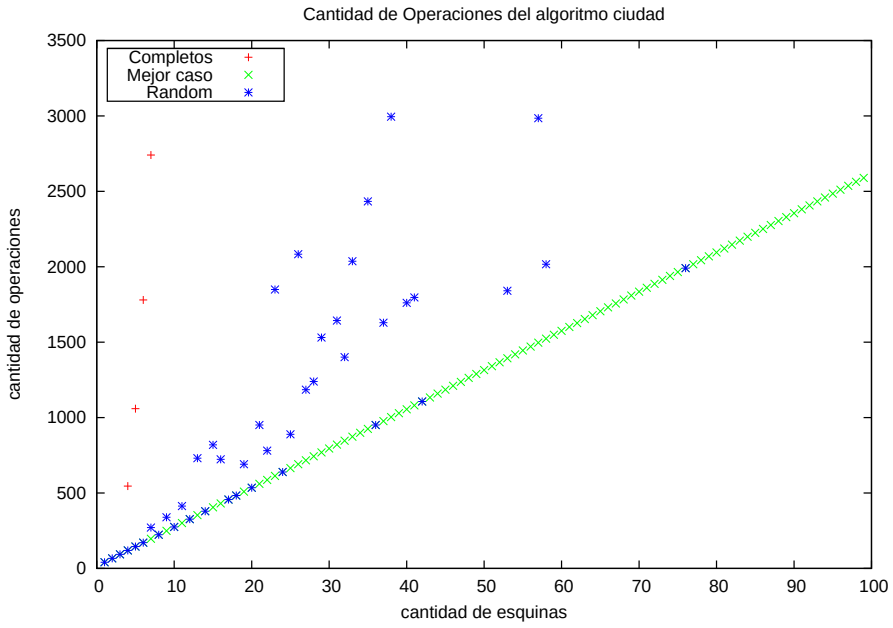
Para contar la cantidad de operaciones realizadas por el algoritmo tenemos un generador de instancias aleatorias, mejor y peor caso. La cantidad de esquinas varía de 0 a 100, siendo *una* la cantidad de instancias para cada n (cabe aclarar que es una instancia aleatoria, una mejor, y una peor).

A continuación se muestra para cada instancia la cantidad de operaciones en función de la cantidad esquinas. Esperamos ver como la función $c * n^3$ acota superiormente nuestro peor caso (con $c = 5{,}8$), y como el mejor acota inferiormente (menor igual) a cualquiera, ya sea aleatoria o peor caso a partir de cierto n.



En el gráfico anterior podemos ver que la complejidad analizada de forma teórica se condice con la cantidad de operaciones realizadas por las instancias generadas. De todas formas no podemos afirmar que el mejor caso es lineal ya que puede estarse tratando de costo constante. Creemos que la escala del eje y, es decir la gran cantidad de valores que puede admitir en un gráfico de ese tamaño, es la que nos da una impresión errónea.

Por lo tanto, el siguiente gráfico mostrará las mismas instancias, pero con un rango de valores del eje y mucho menor. Esperamos ver que en realidad el mejor caso es una recta, y que tanto los casos aleatorios como el peor, se mantienen o igual, o por encima de ella.



Podemos afirmar que el mejor caso se comporta como tal, y tiene costo lineal. Además, verificamos que nuestra hipótesis sobre el comportamiento observado en el primer gráfico es correcta (la escala no era la adecuada).

Vale observar que no tomamos en cuenta el costo de la inicialización de la matriz de adyacencia (su costo es n^2), de esta forma podemos afirmar que nuestro mejor caso para la función ciudad es lineal (la complejidad se analizó sólo sobre la función principal -ciudad-).

2.5. Conclusiones

El modelado del problema mediante un grafo, nos facilitó la resolución del problema. Dejamos de pensar si podíamos asignar direcciones a las calles de una ciudad de manera tal de poder ir de una esquina a cualquier otra, y nos dedicamos a verificar si se cumplía una propiedad en el grafo (esta propiedad es la de ser fuertemente conexo).

La implementación de nuestro algoritmo resultó ser polinomial, gracias a las técnicas conocidas aplicadas a grafos, pudimos mediante un dfs con ligeras modificaciones (las que mantuvieron las bases del algoritmo original), obtener un resultado correcto.

Nos resultó el problema más complicado de resolver, ya que en un principio todas nuestras posibles soluciones no resolvían el problema en todas las instancias posibles y/o no cumplían con la complejidad pedida. La solución final necesito de un crecimiento por parte nuestra, en la forma de ver el problema. Primeramente queríamos buscar ciclos y tratar de unirlos entre sí hasta abarcar todos los vértices. Logramos resolverlo cuando observamos que nuestra primer idea, era la misma que tener un ciclo principal e intentar buscar ciclos conectados a él, ignorando si en el proceso, nos encontrabamos con un ciclo aislado. Aún así, nuestra primer implementación correcta mostraba que necesitabamos todavía refinar el algoritmo. Esta solución fue posible, aplicando la técnica *top down*.

Computacionalmente es un problema 'bien' resuelto, dado que la complejidad resultó ser $O(n^3)$.

3. Problema 3

Bernardo se encuentra en una prisión que consta de n habitaciones conectadas por m pasillos. Cada pasillo conecta exactamente dos habitaciones y puede ser transitado en ambas direciones. En toda la prisión hay p puertas, cada una puede abrirse con una única llave. Tanto las puertas como las llaves están repartidas en las habitaciones, de tal forma que cada habitación puede tener una única puerta o almacenar una única llave, pero no ambas cosas. Si una habitación tiene puerta, la llave correspondiente es necesaria para entrar a la habitación, independientemente del pasillo que se use para llegar a la misma. Bernardo se encuentra en la habitación uno, mientras que desde la habitación n es posible salir de la prisión. Decidir si Bernardo puede recorrer las habitaciones recolectando llaves y abriendo puertas de manera tal de llegar a la habitación n y asi escapar.

3.1. Explicación

Bernardo recorre las habitaciones que tengan conexión con la habitación donde se encuentra, siempre y cuando pueda entrar, ya sea porque tiene la llave o porque la habitación no tiene puerta. Su procedimiento continúa hasta que:

- Llega a la habitación n, por lo tanto encontró la salida.
- Se le terminan los accesos a las habitaciones vecinas, es decir no puede acceder a ninguna habitación todavía no visitada y por lo tanto no puede escapar.

3.2. Detalles de la implementación

Modelamos este problema con un grafo donde los vértices corresponden a las habitaciones (puede ser una habitación con una llave dentro, con una puerta o sin puerta ni llave) y las aristas a los pasillos.

Generamos la matriz de adyacencia del grafo (de $n \times n$ donde n es la cantidad de habitaciones donde cada posición (i, j) de la matriz contiene un *uno* si la habitación i está conectada con la habitación j y *cero* en caso contrario). Nos referiremos a la matriz como *conexiones*.

Para toda habitación que tenga puerta existe una llave. Poseer esta llave implica tener la posibilidad de acceder a la habitación. Podemos abstraernos

del problema de Bernardo y considerar a las llaves como valores booleanos en un arreglo que nos dice para cada vértice, si este es accesible o no. Tenemos entonces, un arreglo de tipo bool ($tengo\_llave$) de tamaño n donde cada vértice, representado por el índice de dicho arreglo, esta seteado en $verdadero$ si es accesible y en $falso$ sino.

Por otro lado, tenemos un arreglo $puertas$ de tamaño n (siendo n es la cantidad de vértices del grafo), donde cada posición, si corresponde a una habitación con llave, tiene el vértice al cual habilita el acceso. En caso contrario, el arreglo contiene el valor $cero$. El valor es $cero$, porque como se verá más adelante en el pseudocódigo, modificará información del primer vértice, que a los fines prácticos, no modifica el resultado final del algoritmo.

También tenemos un arreglo de bools llamado $enEspera$ el cual esta inicializado en falso si el vértice i no puede ser accedido (tiene puerta), y en verdadero en caso contrario. Entonces, para cada vértice, de existir una posible restricción de acceso, su posición permanece en falso.

Para la resolución del problema recorrimos el grafo de forma ordenada por niveles. Para esto hicimos una modificación al algoritmo $Breadth\ First\ Search$. La modificación consiste en visitar los vértices adyacentes al 'actual' tales que todavía no fueron visitados y tienen acceso permitido, cuando esto último no ocurre se pone el vértice en 'espera' hasta que por otro camino se encuentre al vértice que habilite el acceso al mismo. Al momento de conseguir el acceso a un vértice que se encuentra en 'espera' se lo accede directamente (sin volver a pasar por los vértices que llevan a él) considerando posible ese camino hacia el vértice n. Cabe destacar que este algoritmo puede acceder a cada vértice sólo una vez.

El objetivo del bfs es llegar desde el primer vértice al último.

El siguiente pseudocódigo refleja el comportamiento previamente descripto:

```
prision(conexiones, tengo_llave, puertas, n)
    llegue ← false
    cola q
    visitados[0..n] ← false
    enEspera[0..n] ← false
    encolar(q, 0)
    visitados[0] ← true
    while !esVacia(q) and !llegue
        actual ← primero(q)
        desencolar(q)
        for i ← 0 to n and !llegue
            if conexiones[actual][i] and tengo_llave[i] and !visitados[i]
                tengo_llave[puertas[i]] ← true
                encolar(q, i)
                visitados[i] ← true
                llegue ← (i == n − 1)
                if enEspera[puertas[i]]
                    encolar(q, puertas[i])
                    visitados[puertas[i]] ← true
                    llegue ← (n − 1 == puertas[i])
            else
                if conexiones[actual][i] and !visitados[i]
                    enEspera[i] ← true
    return llegue
```

En pocas palabras, la idea del algoritmo es explorar el grafo con bfs en busca de un camino que inicie en el vértice *cero* y llegue al vértice n, si se encuentra con un vértice que no es accesible, espera hasta que desde otro camino se habilite la entrada a ese vértice, una vez habilitada continua por ese camino, mientras sigue recorriendo todo camino que se encuentre habilitado hasta el momento.

El resultado final viene dado por la variable *llegue*. Es inicializada en

$falso$ y se setea en *verdadero* si sólo si en algún momento el vértice a encolar es el n ($i == n - 1$), es decir, el vértice n fue alcanzado por el bfs. Si esto ocurre, podemos concluir que pudimos llegar al vértice objetivo. Si pasa por todos los vértices habilitados y los que pudo habilitar y no puede llegar al vértice n, sale del ciclo (**while**) sin cambiar el valor de *llegue* y devuelve $falso$.

3.3. Análisis de complejidad

Como mencionamos previamente, el algoritmo **no** podrá pasar más de una vez por cada vértice. Mientras quede algun vértice por ver y pueda accederse, seguirá intentando llegar al nodo n. Analizando la complejidad, vemos que si el bfs pudiera acceder indistintamente a todos los vértices del grafo, el ciclo *while* tendría un costo de n iteraciones.

Dentro del ciclo principal tenemos las primeras dos operaciones con costo constante (una asignación y quitar de la pila el primer elemento), y un ciclo anidado (for).

Por lo tanto tenemos hasta el momento n interaciones del ciclo *while* donde dentro de él tenemos un ciclo for. Una primer aproximación a la complejidad final sería pensar que para cada una de las n iteraciones tendremos un costo h todavía no conocido por el ciclo for, descartando para complejidad el costo constante de la asignación y quitar el primer elemento de una pila. Tenemos hasta ahora, $O(n * h)$.

Analizemos ahora la cantidad de operaciones a la que equivale h en el peor caso.

El ciclo for itera en el peor caso desde 0 hasta n (puede salir antes si el valor de la variable bool *llegue* se setea en verdadero). Por lo tanto tenemos que h será a lo sumo $n * c$ porque dentro de este ciclo anidado, se asignan valores a arreglos, matrices, se encolan parámetros constantes a pilas, se setean valores booleanos y se chequea guardas de condicionales if siendo todas estas operaciones, de costo constante. Cabe aclarar que la cantidad de veces que se realizan estas operaciones por cada iteración también es constante, por lo que se puede deducir que una iteración del ciclo for tiene costo constante. Es así entonces que $h = n * c$ (con c constante).

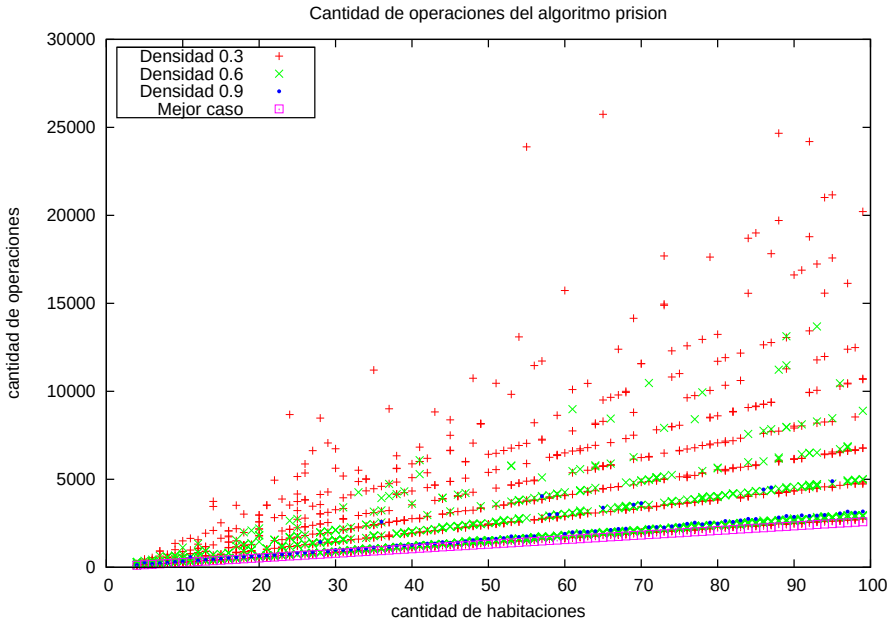
Continuando con el análisis de complejidad, teníamos que el algoritmo tenia complejidad de $O(n * h)$, siendo ahora $h = n * c$. Por lo tanto, la complejidad de este algoritmo es $O(n * n * c) = O(n^2) \subset O(n^3)$

3.4. Pruebas y Resultados

Para probar correctitud contamos con un generador de instancias en las cuales Bernardo puede escapar y otras en las cuales no, que nos devuelve un archivo *test_libre.in* y *test_no.in* respectivamente. De esta forma, con una comparación de archivos (comando `make diffs` en la carpeta Ej3) podemos saber para cada instancia si el resultado obtenido por nuestro algoritmo es correcto.

Para analizar la complejidad temporal del algoritmo (cantidad de operaciones) tenemos un generador aleatorio de 'mapas' donde para cada número de habitaciones n variamos el porcentaje de pasillos con respecto al máximo (es decir, variamos la densidad del grafo). Generamos para cada n desde 4 a 100 (4 por enunciado y 100 porque nos pareció suficente para comprobar el comportamiento del algoritmo), 10 casos con densidad 0,3 y 0,6 y 0,9. También, generamos mejores casos, en donde la habitación 1 no está conectada por ningún pasillo a otra. El algoritmo necesita por lo menos, buscar entre todos los vértices los adyacentes al primero (la habitación de la que sale Bernardo) para poder afirmar que no tiene ninguno y terminar (para toda instacia, el algoritmo necesita de la inicialización de dos arreglos, previo a entrar al ciclo *while*). Deducimos que el costo para estos casos es lineal.

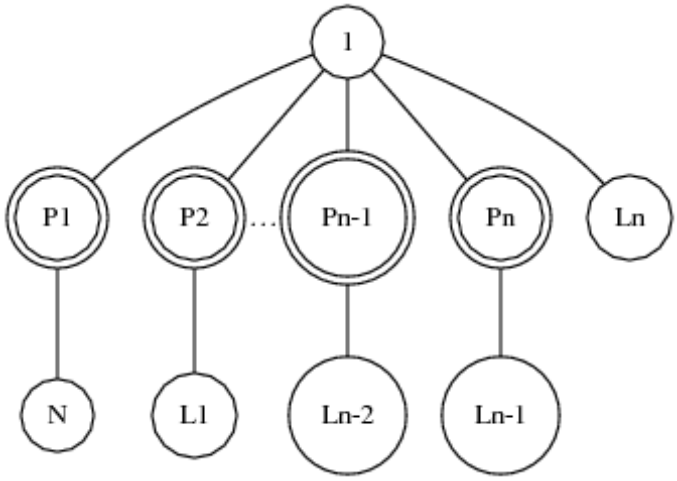
A continuación se muestra para cada instancia la cantidad de operaciones en función de la cantidad de habitaciones y la cantidad de pasillos (densidad del grafo). Esperamos ver que cuánto mayor es la densidad mejor es el algoritmo.



Podemos ver que se cumple lo predicho ya que para un n fijo (para una cantidad de habitaciones fija), la función crece (la cantidad de operaciones aumenta) conforme disminuye la cantidad de pasillos. Además, vemos que cuanto mayor es la densidad más tienden a agruparse, consideramos que esto pasa porque a que a medida que aumenta la densidad la variedad de instancias posibles disminuye, siendo alta la probabilidad de que se repitan al generarlas y siendo las distintas configuraciones del grafo muy similares con respecto al costo.

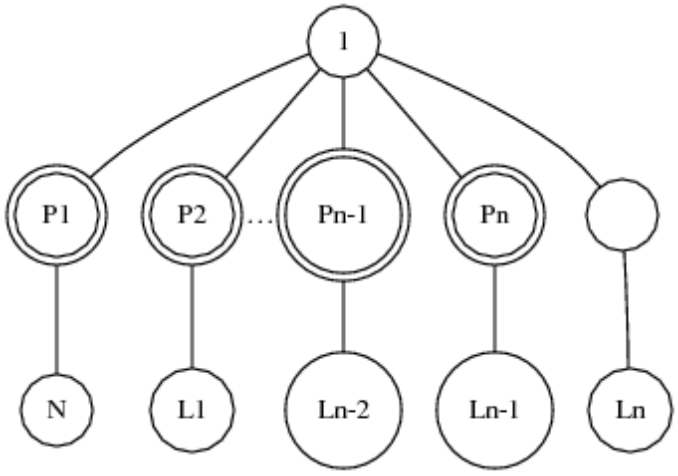
Por otro lado, generamos peores casos para el algoritmo $\forall n$ $(4 \leq n \leq 100)$. Para generarlos, armamos un árbol donde la raíz es la habitación *uno*. La cantidad de llaves/puertas es exactamente $\lfloor \frac{(n-2)}{2} \rfloor$. Si n es par, en el nivel *uno* están todas las habitaciones con puerta y otra que tiene la llave de la habitación $\frac{(n-2)}{2} + 1$. En el nivel *dos* están el resto de las habitaciones, es decir, las habitaciones que tienen las llaves y la última habitación. Si n es impar, en el nivel *uno* están todas las habitaciones con puerta y otra que no tiene llave ni puerta. En el nivel *dos* están el resto de las habitaciones, es decir, las habitaciones que tienen las llaves y la última habitación. En ambos casos, la llave correspondiente a la habitación i, $\forall i(2 \leq i \leq \frac{(n-2)}{2})$ está en la habitación $n - i + 1$. Este tipo de mapa es uno de los peores casos del algoritmo porque para determinar que puede escapar, el bfs visita necesariamente todos los nodos, dejando en espera a todos los bloqueados para luego visitarlos.

La siguiente figura muestra lo descripto anteriormente para n par, siendo la habitación 2 la que contiene la puerta uno:

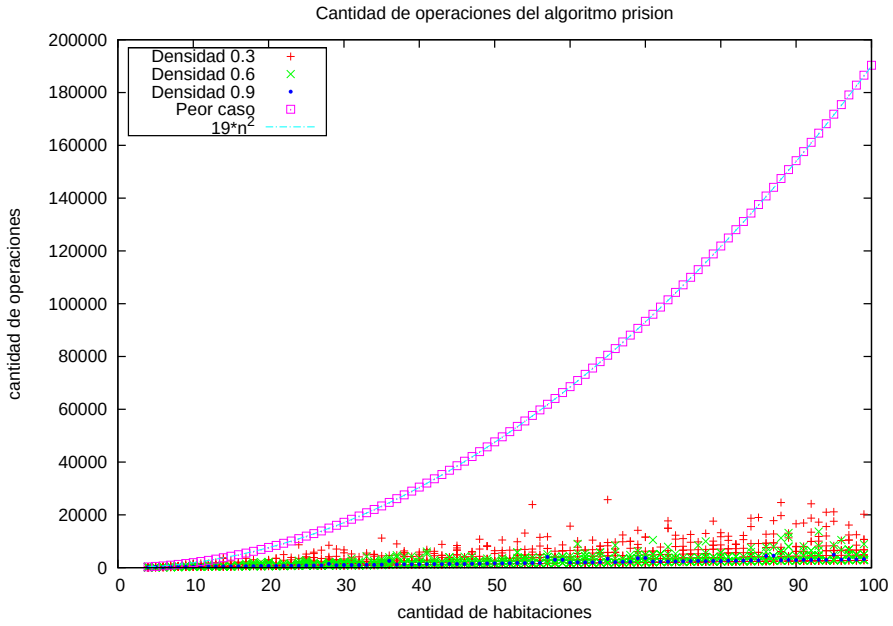


- La letra 'P' se refiere a 'puerta'.
- La letra 'L' se refiere a 'llave'.
- '1' se refiere a la habitación en la que Bernardo inicia.
- 'N' se refiere a la habitación por la que Bernardo podría escapar de la prisión.

La siguiente figura muestra lo descripto anteriormente para n impar, siendo la habitación 2 la que contiene la puerta uno:



En el siguiente gráfico se muestra la cantidad de operaciones en función de la cantidad de habitaciones y la cantidad de pasillos (densidad del grafo). Ademas, graficamos la función $19 * n^2$ ya que es la cota superior previamente calculada con una constante aproximada calculada empíricamente. El objetivo de esta prueba es poder constrastar los casos del gráfico anterior con los peores casos y mejores caso para el algoritmo, para corroborar que las cotas son correctas.



Se puede ver que $19 * n^2$ es la cota superior exacta para el algoritmo. Además, podemos afirmar que las instancias elegidas como peores casos se comportan como tal. En general, vemos que el algoritmo se comporta mucho mejor que la cota. Hicimos 1000 test aleatorios por cada densidad y la cantidad de operaciones no superó las 40000 en ninguno de ellos. En cambio los peores casos superan esta cantidad de operaciones con n a partir de 45 aproximadamente. Podemos concluir que cuanto mayor sea n, mayor será la diferencia de $performance$ entre el peor caso y los elegidos aleatoriamente (con la salvedad de que no se elija un peor caso).

Vemos que la cantidad de operaciones para los mejores casos crece lineal-

mente como habíamos supuesto. Creemos entonces que la cota inferior para el algoritmo fue bien elegida.

Vale observar que no tomamos en cuenta el costo de la inicialización de la matriz de adyacencia (su costo es n^2), de esta forma podemos afirmar que nuestro mejor caso para la función ciudad es lineal (la complejidad se analizó sólo sobre la función principal -ciudad-).

3.5. Conclusiones

Modelar el problema con un grafo, permitió abstraernos de Bernado y la prisión, pudiendo pensar en técnicas para recorrerlos de forma eficiente. Así, realizar un bfs sobre nodos (un algoritmo conocido) con cambios mínimos, satisfizo los requerimientos del problema.

Habiéndolo modelado con grafos y recorrido de forma ordenada, pudimos implementar un algoritmo polinomial de complejidad n^2. Podemos concluir que el problema está computacionalmente 'bien' resuelto.

Al tener muchas variables en el problema, como la cantidad de habitaciones, pasillos, llaves y el lugar donde estas estaban, complejizaron la realización de los gráficos. Terminamos realizando test con cantidad de habitaciones fijas, cantidad de pasillos fijos (la máxima cantidad de pasillos posibles para esa cantidad de habitaciones multiplicaba por una densidad entre 0 y 1) siendo aleatorios los extremos (las habitaciones que conectan), y llaves distribuídas de forma también aleatoria. Creemos que de esta forma, al realizar una cantidad significativa de tests, cubrimos una cantidad suficiente de casos.

4. Mediciones

- Para contar la cantidad aproximada de operaciones definimos una variable inicializada en *cero* la cual incrementamos luego de cada operación. Preferimos contar operaciones en vez de medir tiempo porque a pesar de que es aproximado el resultado, el error es siempre el mismo y así podemos hacer una mejor comparación entre las instancias.
  Midiendo tiempo, el error para cada instancia varía, ya que es el sistema operativo el que ejecuta nuestro programa, al "mismo tiempo"que otras tareas.

5. Compilación y ejecución de los programas

Para compilar los programas se puede usar el comando `make` (Requiere el compilador `g++`). Se pueden correr los programas de cada ejercicio ejecutando `./secuencia_unimodal`, `./ciudad` y `./prision` respectivamente.

Los programas leen la entrada de stdin y escriben la respuesta en stdout. Para leer la entrada de un archivo `Tp1EjX.in` y escribir la respuesta en un archivo `Tp1EjX.out` ses puede usar:
`./(ejecutable) <Tp1EjX.in >Tp1EjX.out`

Para contar la cantidad de operaciones: `./(ejecutable) count`. Devuelve para cada instancia el tamaño seguido de la cantidad de operaciones de cada instancia. En el ejercicio 3 también se puede contar la cantidad de operaciones en función de la cantidad de llaves de la siguiente manera:
`./prision count_llaves`. Devuelve para cada instancia la cantidad de llaves/puertas seguido de la cantidad de operaciones correspondientes.