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 $c_i = \max\limits_{0<j<i}\{c_j / s_j < s_i\} + 1$ $(\forall\ i \in \mathbb{N}, 0 < i \leq n)$, 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 previa 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* 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 $j < i$ ($\forall\ 0 \leq i < n$), 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 max_long_creciente[j] > max_long) O(1)
                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 guardar 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 adentro 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).

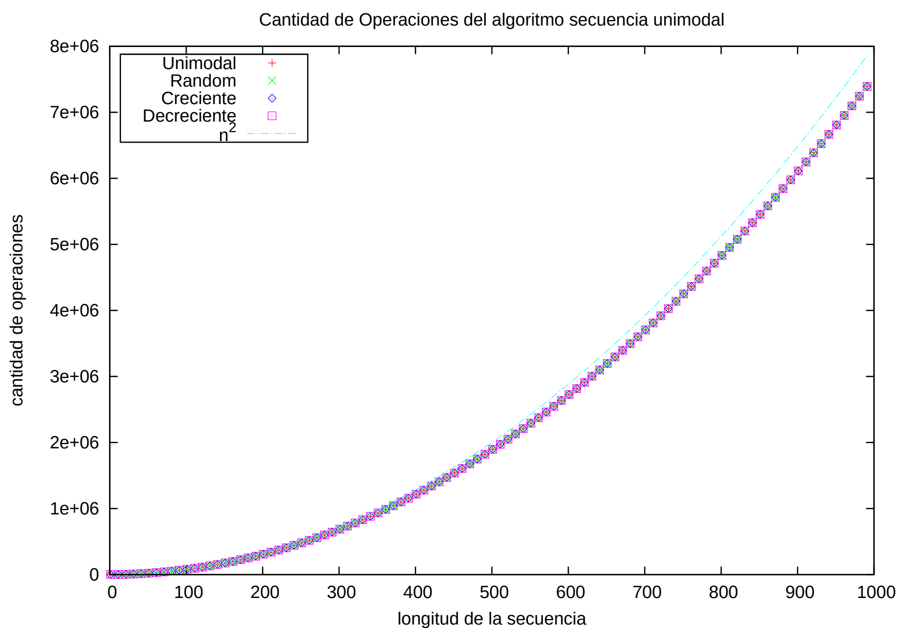
Por lo tanto, el costo del algoritmo *secuencia_unimodal* es: O($2*n+2*n^2$) que por definición es O($max(2 * n, 2 * n^2)$) = O($2 * n^2$) = O(n^2).

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.

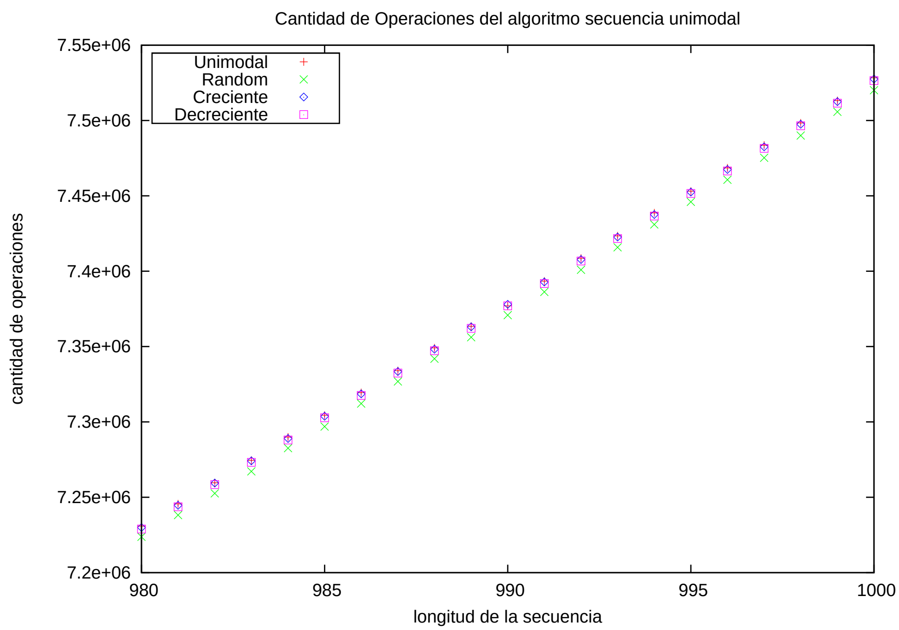
Para llevar a cabo las pruebas de este algoritmo en cuanto a operaciones realizadas por instancia, generamos secuencias unimodales, crecientes, decrecientes y random con el objeto de constatar si los gráficos (a continuación expuestos), se condicen con la complejidad teórica analizada.

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 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, carece de importancia.

2. 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.

2.1. 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 y por lo tanto no puede escapar.

2.2. 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 aritas 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 restriccion 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 $falso$.

2.3. 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 indistitamente 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 todavia 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.

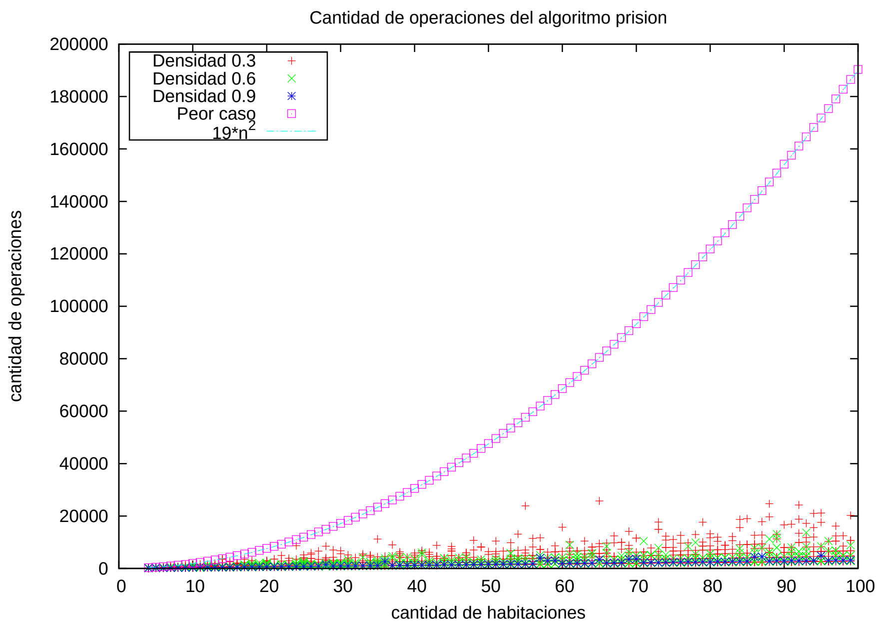
El ciclo for itera en el peor caso desde 0 hasta n (puede salir antes si el valor de la variable bool $llegue$ se setea en verdadero). Por lo tanto tenemos que h será a lo sumo $n*c$ (con c constante) porque dentro de este ciclo anidado, se asignan valores a arreglos, matrices, se encolan parametros constantes a pilas, se setean valores booleanos y se chequea guardas de condicionales if y todas estas operaciones tienen 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 $h = n * c$.

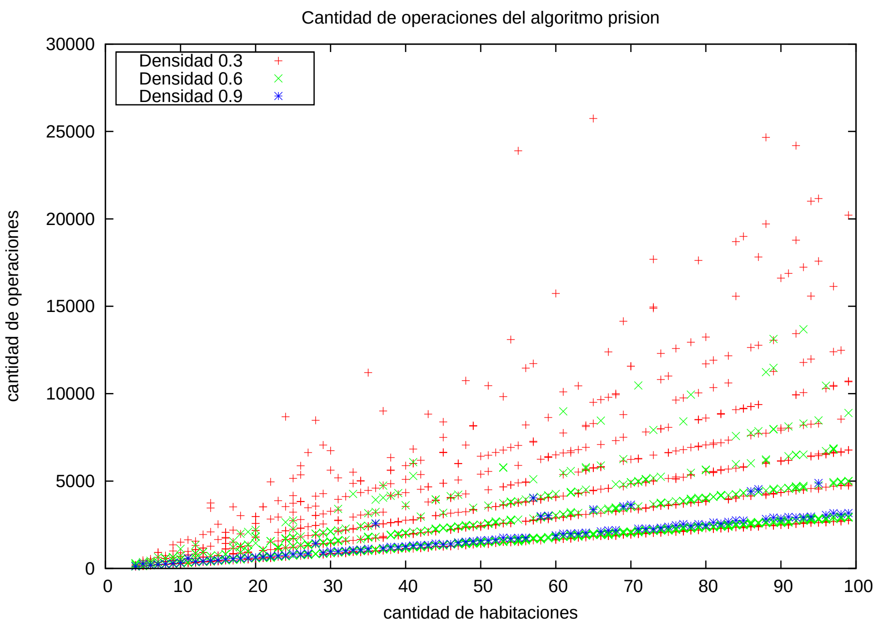
Continuando con el análisis de complejidad, teníamos que el algoritmo tenia complejidad de $O(n*h)$, siendo ahora $h = n*c$. Por lo tanto, la complejidad de este algoritmo es $O(n*n*c) = O(n^2) \subset O(n^3)$

2.4. 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 1 con densidad 0,9, decidimos hacer sólo una instancia de densidad 0,9 por cada n porque la variedad de instancias posibles para esa densidad es baja y la probabilidad de que se repitan alta por lo que nos pareció que no tenia sentido hacer más.


Cantidad de operaciones del algoritmo prision
Densidad 0.3
Densidad 0.6
Densidad 0.9
Peor caso
19*n^2
cantidad de operaciones
cantidad de habitaciones
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
0
10
20
30
40
50
60
70
80
90
100


Cantidad de operaciones del algoritmo prision
Densidad 0.3
Densidad 0.6
Densidad 0.9
cantidad de operaciones
cantidad de habitaciones
30000
25000
20000
15000
10000
5000
0
0
10
20
30
40
50
60
70
80
90
100

3. 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.

4. 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.