DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nro. 1

13/04/2009 Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Dinota, Matías	076/07	matiasgd@gmail.com
Huel, Federico Ariel	329/07	federico.huel@gmail.com
Leveroni, Luciano	360/07	lucianolev@gmail.com
Mosteiro, Agustín	125/07	agustinmosteiro@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

Aclaraciónes generales

Antes de comenzar el análisis de cada ejercicio, cabe mencionar lo siguiente:

- La implementación de los 3 algoritmos se realizó en **lenguaje C++**, haciendo uso de las librerías estándar del mismo.
- Para varios de los experimentos realizados se utilizó un rango de números acotado cuyo máximo valor es $2^{31} - 1$, ya que este es el valor máximo del tipo de datos *int*, utilizado para representar los números naturales y enteros en las implementaciones propuestas.
- Para el cálculo de tiempo de los algoritmos se utilizó la función **gettimeofday()** del estándar POSIX.
- Para verificar la paridad o divisibilidad de un número se utilizó el **operador %** de lenguaje.
- El código fuente de los algoritmos aqui analizados se encuentran en los archivos *encontrarPrimos.cpp* (Ej 1), *puzzleConPoda.cpp* (Ej 2) y *mediana.cpp* (Ej 3).
- El código fuente de los programas encargados de hacer uso de los algoritmos (funciones *main*), junto con la lectura y escritura de los archivos de entrada y salida tal como indica el enunciado se encuentran en los archivos *mainPrimos.cpp* (Ej 1), *mainPuzzle.cpp* (Ej 2) y *mainMediana.cpp* (Ej 3).
- Para la lectura y escritura de los datos se implementó una clase *Archivo* que abstrae las operaciones de manejo de archivo provistas por el lenguaje con el fin de facilitar el uso y la comprensión de los algoritmos. No se hará referencia a estos algoritmos ya que no resultan de interés para el trabajo aqui presentado.
- Los archivos fuente de los programas encargados de generar los datos para la realización de los gráficos, según el ejercicio, son:
  - Ej 1: mainPruebaPrimos1.cpp (Figura 1.1), mainPruebaPrimos2.cpp (Figura 1.2) y mainPruebaPrimos3.cpp (Figura 1.3).
  - Ej 2: mainPruebaPuzzle.cpp
  - Ej 3: mainPruebaMediana.cpp
- Los gráficos se realizaron con **GNUPlot** y las tablas con **OpenOffice Calc**. En los casos considerados pertinentes, se utilizó una escala logarítmica con el fin de poder visualizar mejor los resultados.

Ejercicio 1: La conjetura de Goldbach

Introducción

La idea de este ejercicio es intentar demostrar de manera empírica la conjetura de Goldbach. Esta misma predica que todo número par positivo mayor que dos se puede escribir como la suma de dos números primos. Con este objetivo se procedió a la implementación de un algoritmo iterativo que, dado un número par mayor que dos, retorna dos números primos que satisfacen dicha conjetura.

En principio, se probaron distintos casos aleatorios para encontrar un patrón común. A partir de esto, se realizó la siguiente conjetura: "Sea p el mayor primo comprendido en el rango $3..n-2$ entonces p y $n-p$ son dos números primos cuya suma es n". De ser cierta esta conjetura, bastaría con hallar el primo más cercano a $n-2$ para obtener, mediante operaciones baásicas, el resultado esperado. Sin embargo, al poco tiempo se encontró un contraejemplo ($128 = 119 + 9$, pero 9 no es primo) que falseaba la conjetura. A pesar de esto, la idea sirvió para encontrar una solución al problema. En la sección siguiente veremos el algoritmo en cuestión.

Algoritmo

A continuación se encuentra el pseudocódigo relativo al algoritmo propuesto que resuelve el problema, seguido de una breve explicación acerca del funcionamiento y correctitud del mismo.

```
esPrimo(n)
  si n = 2
    devolver true
  si n es divisible por 2
    devolver false
  para todo i par desde 3 hasta raiz de n
    si n es divisible por i
      devolver false
  devolver true

encontrarPrimos(n)
  si n = 4
    devolver (2,2)
  para todo i par desde 3 hasta n/2
    si esPrimo(i) y esPrimo(n-i)
      devolver (i,n-i)
  devolver (0,0)
```

En primer lugar, se encuentrá la definición de la función *esPrimo* que, tal como su nombre indica, retorna *true* en caso de que el número pasado como parámentro sea un número primo y *false* en caso contrario. El algoritmo en cuestión consiste en un método simple pero efectivo: En primer lugar, hay 2 casos base, el primero retorna *true* en caso de tratarse del número 2 y el segundo evalua si el numero es par (divisible por 2), retornando *false* en este caso, ya que todo numero par mayor que 2 no es primo. Luego, el ciclo principal evalua si el parámetro n es divisible por algún número impar i con $3 \leq i \leq \sqrt{n}$, retornando *false* en tal caso. La correctitud de tal procedimiento se desprende directamente del teorema que predica que si n es un número primo entonces existe al menos un divisor d tal que $2 \leq d < \sqrt{n}$.

Con respecto al algoritmo relativo al problema en sí, la idea consiste en recorrer todos los números impares comprendidos entre 3 y $n-3$ de a pares cuya suma equivalgan a n. Los pares $i, n-i$ satisfacen trivialmente esta condición, ya que $i + (n-i) = n$. Del mismo modo, es fácil ver que para generar todos los números del rango mencionado, basta con variar i tal que $3 \leq i \leq n/2$. De esta forma se ve claramente que esPrimo(i) comprende el rango $3..n/2$, y $esPrimo(n-i)$ el rango $n/2..n-3$. De este modo, cuando se satisface que $esPrimo(i) \& esPrimo(n-i) == true$ se puede afirmar que se ha encontrado el par de números que cumplen con la conjetura: dos números primos cuya suma es n. En caso de que no hallar dicha tupla dentro del rango propuesto, se puede concluir que la conjetura de Goldbach era falsa, retornando la tupla (0,0) en este caso.

Complejidad

A continuación se analizara la complejidad de peor caso sobre los modelos uniforme y logaritmico.

Nota: En el siguiente análisis se hara referencia a funciones propias de la implementación de los algoritmos. Puede ver el código fuente del mismo en la sección *Anexos*.

Modelo Uniforme:

En este modelo el análisis no esta centrado en el tamaño de los operandos por lo que el tiempo de ejecución de cada operación se considera constante.

En la función *esPrimo* todas las operaciones y asignaciones utilizadas se logran en tiempo constante por lo mencinado anteriormente, de esta manera la complejidad de esta función depende

del ciclo que contiene. Como itera sobre i entre 3 (valor de inicialización) y $\sqrt{n}+1$, y ademas en cada vuelta del ciclo la variable es aumentada en 2, claramente se aprecia que en el peor caso (es decir que nunca se cumpla la condición n mod i = 0) el ciclo se recorre aproximadamente $\sqrt{n}/2$ veces. De esta manera concluimos que $T(n) \in O(\sqrt{n})$.

En cuanto a la complejidad de las operaciones y asignaciones realizadas en la función *encontrarPrimos* el caso es el mismo al de *esPrimo* debido al modelo de computo. La unica operación dentro del pseudocodigo de encontrarPrimos de complejidad superior a constante es la analizada anteriormente y dicha operacion se encuentra dentro del único ciclo de la función por lo que concentraremos el análisis de complejidad aqui. En este caso iteramos sobre el mismo valor de inicio pero el limite del ciclo esta dado por $n/2$. Tambien aqui la variable i es aumentada en 2 en cada iteración lo que determina que en el peor caso se iterará aproximada mente $(n/2)/2 = n/4$ veces. Como dentro del ciclo se llama a esPrimo (complejidad $\sqrt{n}$) la complejidad final de la función encontrarPrimos es $T(n) \in O(n * \sqrt{n})$.

Modelo Logarítmico:

En este modelo de computo si nos concentraremos en el tamaño de la entrada y el costo de las operaciones en función de la cantidad de bits de sus operandos.

En la función *esPrimo* se puede apreciar que las operaciónes que mayor complejidad conllevan son $\sqrt{n}$ y n mod i. Como ambas son llamadas dentro del único ciclo de la función, la complejidad estará determinada por dicho ciclo.

Entonces tenemos

$$\begin{aligned} T(n) &= \sum_{i=3, i\,mod\,2=1}^{\sqrt{n}} [\log(\sqrt{n}+1) + \log(n)^2 + \log(n)^2 + \log(n\,mod\,i) + \log(i)] \\ &\leq \sum_{i=1}^{\sqrt{n}} [\log(n) + 2\log(n)^2 + \log(n) + \log(n)] \\ &= \sqrt{n}(3\log(n) + 2\log(n)^2) \Longrightarrow T(n) \in O(\sqrt{n} * \log(n)^2) \end{aligned}$$

Con respecto a la función *encontrarPrimos* sucede lo mismo con respecto a que las operaciones de mayor complejidad son realizadas dentro del ciclo por lo que en este caso la complejidad total de la función también estará determinada por dicho ciclo.

La función de complejidad es

$$\begin{aligned} T(n) &= \sum_{i=3, i\,mod\,2=1}^{n/2} [\log(i) + \sqrt{i} * \log(i)^2 + \sqrt{n-i} * \log(n-i)^2 + \log(n) + \log(n) + \log(i)] \\ &\leq \sum_{i=1}^{n} [\log(n) + \sqrt{n} * \log(n)^2 + \sqrt{n} * \log(n)^2 + \log(n) + \log(n) + \log(n)] \\ &= n * (4 * \log(n) + 2 * \sqrt{n} log(n)^2) \Longrightarrow T(n) \in O(n^{3/2} * \log(n)^2) \end{aligned}$$

Por último analizaremos también la complejidad en función del tamaño de la entrada para los dos modelos de cómputo vistos. Como la función encontrarPrimos toma unicamente un natural n, si llamamos t al tamaño de entrada tenemos que $t = \log(n) \Longrightarrow 2^t = n$. De esta manera, reemplazando n en las complejidades calculadas previamente obtendremos el costo de los dos algoritmos.

- Modelo Uniforme:

$$T(n) \in O(n\sqrt{n}) = O(n^{3/2}) \Longrightarrow T(t) = O((2^t)^{3/2}) = O(2^{3t/2})$$

- Modelo Logarítmico:

$$T(n) \in O(n^{3/2}\log(n)^2) \Longrightarrow T(t) = O((2^t)^{3/2}(\log(2^t))^2) = O((2^{3t/2}t^2)$$

Análisis de resultados

Con el fin de analizar la eficiencia del algoritmo propuesto se realizaron diversas pruebas. En primer lugar, se hicieron experimentos orientados a estudiar el tiempo de ejecución del mismo en relación al parámetro ingresado (de ahora en adelante, Prueba 1). Para tal propósito se procedió a la creación de un programa de prueba (ver pseudocódigo debajo) que calcula el tiempo de ejecución de la función *encontrarPrimos* para valores entre 4 y $2^{31} - 4$. Al tratarse de una cantidad de números tan grande, como veremos más adelante, la ejecución real de dicha prueba demoraría en exceso (años). Por tal motivo se decidió tomar números pares separados por una distancia aleatoria entre 10^4 y $2 * 10^4$. La razón del uso de un parámetro aleatorio radica en poder conseguir una buena muestra representativa del rango propuesto anteriormente y evitando así casos patológicos.

```
Prueba 1
--------
mientras i <= 2^31-4
  T = tiempo que tarda la operacion encontrarPrimos(i) en microsegundos
  i = i + 10000 + numero aleatorio par entre 0 y 10000
  guardar en Ej1-Complejidad.txt la linea "i T"
```

La segunda prueba tiene como objetivo mostrar que la cantidad de operaciones que realiza el algoritmo es significativa menor que la cota teorica propuestas, aún en los peores casos. Con este fin, se creó una aplicación que obtiene los máximos de la primer componente de la tupla solución (número primo mas chico) para 20 ejecuciones del algoritmo. Al igual que en la primer prueba, se optó por un enfoque aleatorio, tomando muestras de nḿeros a intervalos aleatorios entre 10^5 y $2 * 10^5$.

```
Prueba 2
--------
para j desde 0 hasta 20
  peor_caso_primo1 = 0
  mientras i <= 2^31-4
    <primo1, primo2> = encontrarPrimos(i)

    si primo1 > peor_caso_primo1
      peor_caso = i
      peor_caso_primo1 = primo1
      peor_caso_primo2 = primo2

    i = i + 100000 + numero aleatorio par entre 0 y 100000

  guardar en Ej1-MayoresPrimos.txt la linea "peor_caso peor_caso_primo1 peor_caso_primo2"
```

Por último, con el fin de observar más en detalle el comportamiento del algoritmo se decidió extraer los peores casos de la Prueba 1 y estudiarlos en profundidad (Prueba 3). Para realizar esta tarea, se implementó una sencilla aplicación (ver pseudocódigo debajo) que toma alrededor de 200 secuencias consecutivas de 1000 números aleatorios comprendidas entre 4 y $2^{31} - 4$ utilizando el mismo criterio que en el caso anterior. Luego, obtiene el máximo tiempo de ejecución entre todos los números de cada secuencia.

```
Prueba 3
```

```
--------
j = 1
mientras i <= 2^31-4
  maximo_tiempo = 0
  T = tiempo que tarda la operacion encontrarPrimos(i) en microsegundos

  si T > maximo_tiempo
    maximo_tiempo = T
    peor_caso = i

  si j = 1000
    guardar en Ej1-Peorescasos.txt la linea "peor_caso maximo_tiempo"

  j = j + 1
  i = i + 10000 + numero aleatorio par entre 0 y 10000
```

A continuación se presentan los resultados relativos a la primera aplicación (Prueba 1).

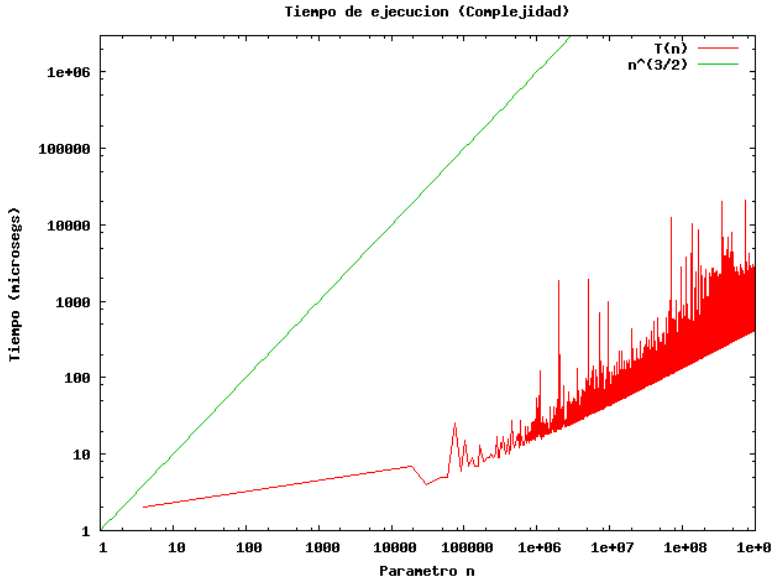

Figura 1.1

Este gráfico muestra la comparación entre las mediciones de tiempo de la primera prueba realizada y la complejidad teórica expuesta anteriormente. Llamaremos f a la función que relaciona el tiempo de ejecución con el tamaño de entrada. Como se puede apreciar, el crecimiento de f es mucho menor que la complejidad teórica, por lo cual el algoritmo resulta mucho más eficiente que lo esperado. Sin embargo, a pesar de su comportamiento altamente irregular, se puede apreciar que el tiempo de ejecución aumenta a medida que crece el tamaño de la entrada. La razón de la discrepancia entre ambas funciones reside en el hecho de que el algoritmo implementado encuentra dos números primos que cumplen con la conjetura en una muy baja cantidad de operaciones en relación al tamanño de la entrada. Si bien se desconoce el motivo de este fenómeno, hemos probado empíricamente que esto se cumple como se puede notar en la siguiente tabla (Prueba 2).

Parámetro n	Primo p1	Primo p2
951.328.178	601	951.327.577
1.465.042.658	571	1.465.042.087
2.021.808.872	739	2.021.808.133
946.027.838	661	946.027.177
488.898.002	643	488.897.359
427.969.778	619	427.969.159
1.405.023.476	547	1.405.022.929
1.935.768.242	613	1.935.767.629
347.574.664	641	347.574.023
1.978.837.538	661	1.978.836.877
1.892.835.848	601	1.892.835.247
1.032.356.308	701	1.032.355.607
1.480.297.292	601	1.480.296.691
1.714.747.588	509	1.714.747.079
1.089.509.332	719	1.089.508.613
1.311.996.866	619	1.311.996.247
1.997.154.638	607	1.997.154.031
159.798.208	587	159.797.621
1.721.150.836	557	1.721.150.279
1.660.057.508	709	1.660.056.799

Figura 1.2

Como se observa en la Figura 1.2, para los peores casos la diferencia entre cada uno de los primos solución es muy alta, muy cercana al parámetro de entrada n. Esto explica que la complejidad teórica no se corresponda con el tiempo de ejecución.

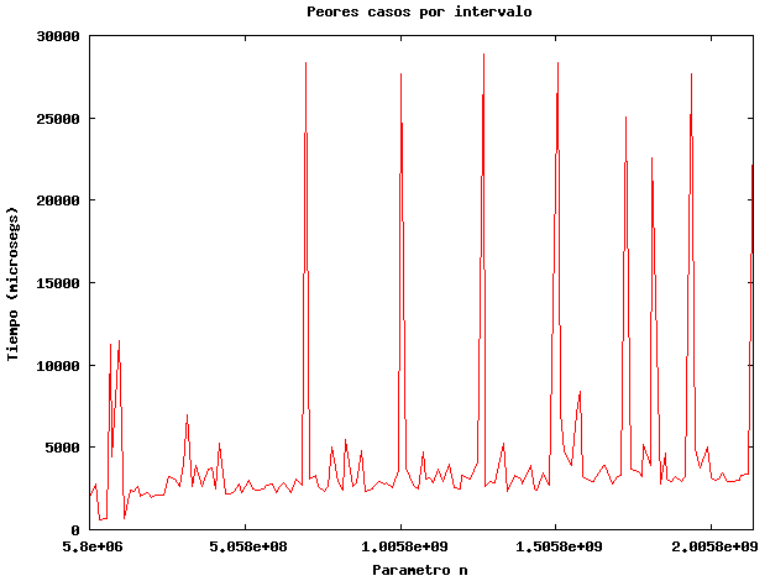

Figura 1.3

La idea de este gráfico es mostrar cúan irregular es el tiempo de ejecución del algoritmo. A pesar de haber hecho el esfuerzo de buscar los peores casos (mediante la aplicación de la Prueba 3), el tiempo que demoran los mismos difieren entre ellos considerablemente. Esto muestra que los peores casos no tienen una ralación directa con el tamaño de la entrada sino mas bien con la distribución desconocida de los números primos pertenecientes al intervalo estudiado.

Conclusiones

Este ejercicio nos mostró un caso representativo en el que el tiempo de ejecución difiere notablemente a la complejidad calculada para el peor caso.

Para este algoritmo en particular se debe a que la función *encontrarPrimos* alcanzaría dicho peor caso si la conjetura de Goldbach fuese falsa, cuestión que no sucede para ningún número que pueda tomar la funcón (es decir, hasta el número mas grande almacenable en un *int*). Más aun, esta demostrado que para números mucho mas grandes que los testeados la conjetura se sigue cumpliendo (ver sección Referencias para más detalles) por lo que en el ámbito científico se cree que es cierta.

También vimos casos en los que entre números cercanos tomados por la función, el tiempo de ejecución difiere considerablemente. Además notamos que a medida que el tamaño del parámetro de entrada crece el tiempo tambíen lo hace, pero siempre muy lejos de los peores casos para dichas entradas. Estas situaciones se deben a la desconocida distribución de los números primos entre los naturales, por lo que no podemos dar una explicación general que incluya todos los casos reales. Sin embargo, en base a todos los análisis realizados, podemos concluir que para todos los casos computables por nuestro algoritmo el tiempo de ejecución será mas que aceptable.

Ejercicio 2: Resolución de Puzzle

Introducción

El objetivo del siguiente ejercicio es disear un algoritmo que, dado un tablero de tamaño n y n^2 fichas, determinar si existe una manera de colocar las fichas en el tablero, y en tal caso, retornarla. La condición que determina si una ficha puede ser colocada es que, en cada uno de sus lados, contenga el mismo número que tiene la ficha con la que limita. Vale aclarar que las fichas no pueden ser rotadas y que los que lados limitan con un borde pueden tener cualquier número.

Cuando empezamos a pensar el ejercicio surgió la solución trivial, tan simple como ineficiente, utilizando un algoritmo de fuerza bruta que prueba todas las permutaciones posibles de las fichas en el tablero. Rápidamente se descartó esta opción pues su costo es factorial. Se optó entonces por aplicar la idea de *backtracking* en la función, de esta manera, se toma una ficha del conjunto restante y se chequea si puede ser colocada en la próxima posición. En caso afirmativo, la ficha se coloca en dicha posición y se realiza el mismo procedimiento para la siguiente casilla del tablero, sino se sigue buscando hasta encontrar una ficha que pueda ser colocada. Si ninguna de las fichas restantes puede ser colocada en el tablero entonces se regresa a la posición anterior (aquí queda explicita la idea de *backtracking*) y se prueba con otra. Por último, se planteó el uso de diversas podas para mejorar el rendimiento del algoritmo. Se dejaron de lado distintas opciones, ya sea por dificultad de implementación o exceso de complejidad temporal o espacial, tomando finalmente la poda que mejor se ajustó a los requerimientos necesarios.

Algoritmo

A continuación se presenta el pseudocódigo del algoritmo para resolver el problema propuesto.

```
resolverPuzzle(fichas[0..n^2-1], n)

  tablero[n][n]

  fichaActualEn[n][n]
  para i desde 0 hasta n-1
    para j desde 0 hasta n-1
      fichaActualEn[i][j] = 0

  fichasPuestas[0..n^2-1]
  para i desde 0 hasta n^2-1
```

```
  fichasPuestas[i] = false

i = 0
j = 0

lista jardineroViejo = lista vacia
lista jardineroNuevo = lista vacia
metio = false
volviolgunavez = false
i_ant = 0
j_ant = 0

mientras i < n

  si !fichasPuestas[fichaActualEn[i][j]]
  y encaja(fichas[fichaActualEn[i][j]], tablero, i, j)

    si i != n-1
      vaciar(jardineroNuevo)
      para k desde 0 hasta n^2-1
        si arriba(fichas[k]) = abajo(fichas[fichaActualEn[i][j]]) y !fichasPuestas[k]
          jardineroNuevo = jardineroNuevo U fichas[k]

    si j = 0 o i = n-1 o !puedoPodar(jardineroViejo, jardineroNuevo)

      si i != n-1
        jardineroViejo = jardineroNuevo

      tablero[i][j] = fichas[fichaActualEn[i][j]]
      fichasPuestas[fichaActualEn[i][j]] = true
      metio = true
      si j = n-1
        i = i + 1
        j = 0;
      sino
        j = j + 1

  si !metio

    volviolgunavez = false;
    mientras fichaActualEn[i][j] = n^2-1

      si i = 0 y j = 0
        devolver "No hay solucion"
      fichaActualEn[i][j] = 0
      si j = 0
        i = i - 1
        j = n-1
      sino
        j = j - 1

      fichasPuestas[fichaActualEn[i][j]] = false
      volviolgunavez = true
```

```
      si volviolgunavez = true
        si j = 0
          i_ant = i-1
          j_ant = n-1
        sino
          i_ant = i
          j_ant = j-1

        si (j_ant != 0 o i_ant != 0) y (i_ant != n-1)
          vaciar(jardineroViejo)
          para k desde 0 hasta n^2-1
            si arriba(fichas[k]) = abajo(fichas[fichaActualEn[i_ant][j_ant]])
            y !fichasPuestas[k]
              jardineroViejo = jardineroViejo U fichas[k]

      fichaActualEn[i][j] = fichaActualEn[i][j] + 1

  devolver tablero


puedoPodar(lista1, lista2)
  devolver interseccion(lista1, lista2) != vacio

encaja(ficha, tablero, i, j)
  (i = 0 o abajo(tablero[i-1][j]) = arriba(ficha)) y
  (j = 0 o derecha(tablero[i][j-1]) = izquierda(ficha))
```

Antes de comenzar, para la implementación se utilizaron las siguientes estructuras:

- **tablero:** es la matriz en donde se irán colocando las fichas que son posible solución hasta un paso determinado. Fue implementada mediante un arreglo de arreglos.
- **fichas:** es el arreglo que contiene todas las fichas de entrada del algoritmo (provenientes de Tp1Ej2.in).
- **fichaActualEn:** cada posición de esta matriz contiene la posición en el arreglo fichas de la ficha actual (la ficha que está siendo evaluada como solución del tablero en un paso del algoritmo) para cada una de las casillas del tablero. Al igual que tablero, esta estructura está implementada con un arreglo de arreglos.
- **fichasPuestas:** es un arreglo de datos *booleanos* del mismo tamaño que fichas, que indica si una determinada ficha está colocada en el tablero.
- **jardineroViejo/Nuevo:** estructura implementada a través de una lista enlazada, que se utiliza, como veremos más adelante, para aplicar las podas del algoritmo.

Luego de inicializar las estructuras, comienza el ciclo principal que intentará colocar las fichas de una en una en el tablero finalizando una vez que este se complete (es decir, cuando $i = n$). En primer lugar se verifica que la ficha actual, apuntada por fichaActualEn[i][j], encaje en la posición (i,j) del tablero y que la misma no pertenezca al conjunto de fichas puestas. En caso afirmativo, se procede a evaluar si se puede realizar una poda (posteriormente explicada en la sección Poda). De no ser factible esta última, se coloca la ficha en el tablero y se avanza a la siguiente posición del mismo. Si se llega a aplicar la poda, la ficha no se inserta.

En caso de que no se haya insertado una ficha en algún paso del algoritmo, se intentará colocar la siguiente ficha (es decir, se aumenta fichaActualEn[i][j]). Si la ficha revisada resulta ser la

última por revisar, se retrocede hasta la posición del tablero más cercana que tenga fichas por revisar aún. Como se sugirió anteriormente, dicha comprobación se realiza por medio de la matriz fichaActualEn. Si se intentara retroceder de la posición inicial (0,0) la instancia recibida no tiene solución y, por lo tanto, el algoritmo finaliza informando tal evento.

Poda

Con el fin de optimizar el rendimiento del algoritmo se aplic el concepto de *poda* visto en clase, relativo a todo algoritmo de *backtracking*. La poda propuesta consiste en la siguiente idea que veremos a continuación. Cabe aclarar que esta poda no se aplica a los bordes izquierdo e inferior del tablero, ya que idea en cuestión no es aplicable en dichas posiciónes.

En primer lugar, cada vez que se desea aadir una ficha, se genera un conjunto determinado de fichas relacionadas a la misma que llamaremos "jardinero". Un "jardinero" está conformado por todas las fichas que pueden unirse por debajo con la ficha a la cual está asociada el jardinero. . Como se puede ver en el pseudocódigo, una vez que se verificó que la ficha actual es candidata para ser insertada en el tablero (aquella que encaja en la posición actual según lo visto antes) se crea el jardinero asociado a esa ficha que denominaremos *jardineroNuevo*. En el caso de encontrarse en el borde izquierdo del tablero, la poda no se aplica y simplemente *jardineroNuevo* pasa a ser *jardineroViejo*. Este último siempre va a ser el jardinero asociado a la ficha anterior a la posición actual del tablero.

Para los casos no pertenecientes al borde izquierdo (ni inferior), se evalua si se puede realizar la poda mediante la funcíon *puedoPodar(jardineroViejo, jardineroNuevo)*. Tal como muestra el pseudocódigo, la misma verifica si hay intersección entre ambos jardineros, es decir, si alguna de las fichas de un jardinero puede encajar con alguna ficha del otro. Si ninguna encaja, implica que la ficha actual (*fichas[fichaActualEn[i][j]]*) que se estaba por insertar no puede ir junto con la ficha anterior (ficha a su izquierda) en esa ubicación del tablero (quizás si pueden formar parte del borde inferior). Observando este hecho desde el punto de vista de un algoritmo de *backtracking*, al descartar esta posible solución se evita tener que ver todas las soluciones que involucren el haber insertado tal ficha en el tablero, evitando así tener que analizar esa "rama" del árbol k-ario de posibles soluciones. Por este motivo, podemos afirmar que este procedimiento realizado constituye una *poda* del árbol de *backtracking*.

Para poder mantener el invariante de tener siempre un jardinero asociado a la ficha anterior a la actual (*jardineroViejo*) es necesario contemplar el caso en el que se retrocede a posiciones anteriores del tablero y se retira fichas insertadas anteriormente. En tal caso, como se puede ver en el pseudocódigo, se genera un *jardineroViejo* que se asocia la posición anterior a la actual (es decir, la posición (i_ant, j_ant)). De este modo, al iterar nuevamente, el jardinero de la nueva ficha a insertar (jardineroNuevo) puede evaluarse con el jardineroViejo para ver si se puede realizar una nueva poda.

Como veremos enseguida, esta poda resulta efectiva en la práctica, disminuyendo notoriamente los tiempos de ejecución.

Complejidad

Como el algoritmo propuesto se basa en la técnica de *backtracking*, la complejidad del mismo en el peor caso es similar a la de un algoritmo de fuerza bruta. En este tipo de algoritmos, se generan todas las soluciones posibles y se evalúan para determinar si cumplen con el problema a resolver. La complejidad de generar y revisar todas las soluciones posibles es exponencial y, en particular, el algoritmo de fuerza bruta aplicado al problema presentado sería $O(n!)$, con n la cantidad de fichas del tablero. A pesar de que los algoritmos de *backtracking* descartan gran cantidad de casos y aplican podas para reducir el tiempo de ejecución, existen casos patológicos en que pocas de estas optimizaciones pueden ser utilizadas y, por lo tanto, se recorran todas las ramas de soluciones. Estos casos hacen que la complejidad en el peor caso del algoritmo presentado sea la misma que la de un algoritmo de fuerza bruta, es decir $O(n!)$ (siendo n la cantidad de fichas totales del tablero).

Sin embargo, como estudiaremos en posteriores secciones, el tiempo de ejecución del algoritmo difiere significativamente del tiempo del algoritmo de fuerza bruta.

Análisis de resultados

En la siguiente sección se mostrarán los análisis realizados para tableros de distinto tamaño. El enfoque principal de las pruebas radica en estudiar los tiempos de ejecución del algoritmo con y sin la poda. Para esto se implementó un algoritmo que genera tableros aleatorios de un tamaño n pasado como parámetro. De esta manera, utilizando dicho algoritmo para distintos valores de n, se contrastaron ambos metodos (con y sin poda) con la cota estudiada ($n!$) arrojando los siguientes resultados. Vale aclarar que se utilizó una escala logarítmica sobre el eje de ordenadas ya que los tiempos de ejecución toman valores muy altos, además, para realizar los gráficos expuestos, se utilizaron tableros hasta un tamaño de $n = 15$ pues para mayores valores los tiempos son excesivos.

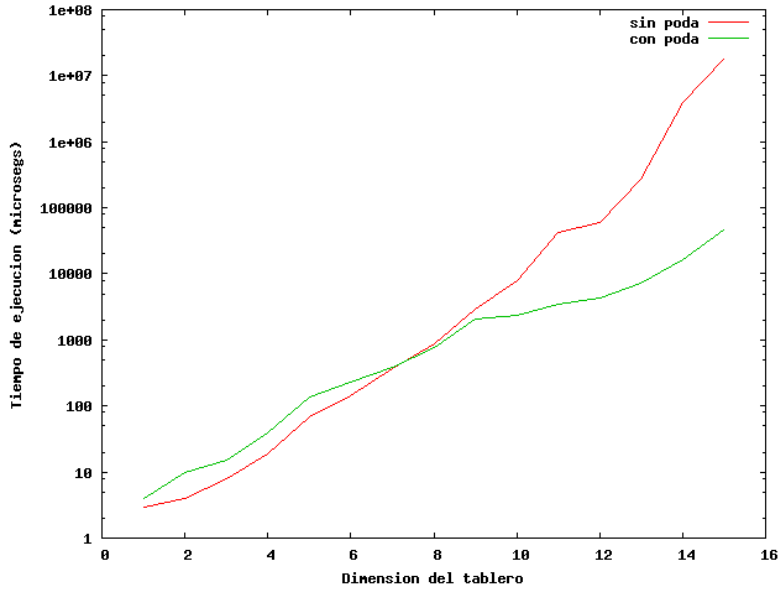

Figura 2.1

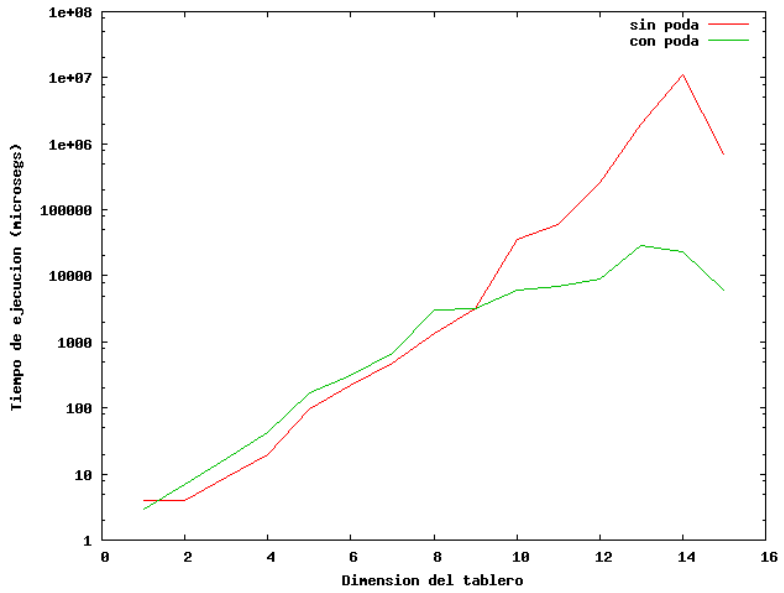

Figura 2.2

Como se puede apreciar en ambos gráficos, los tiempos de ejecución de los dos algoritmos son sensiblemente menores que los de la complejidad teorica calculada para el peor caso ($O(n!)$). Esto concuerda con lo esperado ya que como se dijo en anteriores secciones, salvo en casos patológicos, en la práctica el metodo de *backtracking* aplicado a este problema es mucho más eficiente.

Con respecto a las diferentes resultados obtenidos con los dos métodos, vemos reflejado que efectivamente la poda produce una importante mejora en el rendimiento. Si bien para tableros pequeños el rendimiento es similar (es más, en la mayoría de los casos para n chicos el método que utiliza la poda suele ser menos eficaz), esto no es de importancia ya que en dichos casos, al no manejar gran cantidad de datos, el tiempo de ejecución no alcanza valores muy altos. Por el contrario, cuando tomamos valores de n mas grandes (aprox $n \geq 10$), se puede apreciar que la poda descarta gran cantidad de soluciones no posibles, lo que incide directamente en la eficiencia de la aplicación. De hecho, cuando se analizaron tableros de tamaño mayor que 15, el tiempo que llevó a cabo el algoritmo sin la poda para buscar una solución resultó excesivo, situación que no ocurre cuando se utiliza la poda para tableros de hasta $n = 25$.

Por último, cabe notar que el tiempo de ejecución real no está estrictamente relacionado con el tamaño de entrada. Podemos apreciarlo en la *Figura 2.2*, donde la ejecución de el algoritmo con y sin la poda tomó más tiempo para $n = 14$ que para $n = 15$. Esto se debe a que, además de depender de la dimensión del tablero, la eficiencia del algoritmo estará relacionada con la distribución de las fichas recibidas.

Conclusiones

A lo largo del desarrollo del ejercicio, se experimentaron las particularidades de un algoritmo de *backtracking*. Nos encontramos con diversas dificultades en la implementación, a pesar de que la idea intuitiva del algoritmo es relativamente simple. Además, llegamos a la conclusión de que, si bien la complejidad en peor caso es muy elevada, el algoritmo en general suele ser mucho más eficiente que algoritmos que utilizan la técnica de fuerza bruta para resolver este tipo de problemas.

Los principales incovenientes que surgen al implementar un algoritmo de *backtracking* tienen que ver con la elección de la poda. Al momento de dicha elección, se deben considerar los aspectos de eficiencia, dificultad en la implementación y sobre todo, los beneficios (en relación al tiempo de ejecución) que esta provocaría. Al pensar en detalle cada uno de las posibles podas, estos factores fueron determinantes para tomar una decisión; podas que parecían muy eficientes en principio, resultaban muy difíciles de implementar y no aportaban cambios significativos que justifiquen su uso. A su vez, podas que parecían descartar muchos casos, terminaban empeorando el tiempo de ejecución.

Es importante destacar que al aplicar la poda propuesta, el algoritmo mejora considerablemente en promedio. Esta situación resultaba esperable y pudo ser comprobada empíricamente en las pruebas realizadas. También, se notó una relación entre la distribución de fichas de entrada y el tiempo de ejecución del algoritmo presentado. Por ejemplo, existen casos en los que la disposición de las fichas no permitan hacer podas y otros en los que no sea necesario tener que reemplazar alguna ficha.

En conclusión, la eficiencia del algoritmo de *backtracking* depende fuertemente de las podas que se realicen y de la distribución probabilística de los parámetros de entrada para casos promedio. Por esta razón, la implementación de podas es determinante y justifica el tiempo invertido en idear las mismas.

Ejercicio 3: Cálculo de la mediana entre dos vectores

Introducción

El problema que plantea este ejercicio es calcular la mediana entre dos vectores de números enteros, es decir, el elemento que, una vez concatenados y ordenados los vectores, deja a ambos lados la misma cantidad de elementos. Según lo especificado en el problema, ambos vectores tienen la misma longitud y están ordenados.

En principio, la solución trivial del problema fue descartada ya que el costo de realizarla era excesivo para los fines del ejercicio. Esta solución consistía en concatenar ambos vectores, ordenarlos y posteriormente obtener la mediana a partir del elemento medio de dicha secuencia.

Descartada esta opción, surgió la idea de desechar una gran cantidad de elementos, cuando sea posible, para poder realizar los cálculos sobre una cantidad de números considerablemente inferior a la inicial. De esta manera, la complejidad resultaría del costo de descartar los elementos apropiados y luego hacer el cálculo con una cantidad pequeña de elementos. Evidentemente, nuestro algoritmo seguiría el concepto de Divide and Conquer clásico.

Con las ideas más claras, notamos que descartando la misma cantidad de elementos mayores o iguales a la mediana, que menores o iguales a ella, ésta se mantenía como valor medio de los vectores generados a partir de los elementos no descartados. Rápidamente, nos dimos cuenta que la mediana de concatenar dos vectores ordenados, se encuentra entre los elementos mayores a la mediana del vector con menor valor medio y los elementos menores a la mediana del vector con mayor valor medio (conjetura que luego se demostrará).

Planteado todo esto, resultó sencillo escribir el pseudocódigo de nuestro algoritmo.

Algoritmo

A continuación se mostrará el pseudocódigo del algoritmo implementado seguido de una demostración de su correctitud.

```
medianaRecursiva (vector x[0..n-1], vector y[0..n-1])

  si n == 1
    si x[0] > y[0]
      devolver y[0]
    sino
      devolver x[0]

  si n == 2
    si x[0] == y[0]
      devolver x[0]
    si x[0] > y[0]
      si x[0] > y[1]
        devolver y[1]
      sino
        devolver x[0]
    sino
      si x[1] > y[0]
        devolver y[0]
      sino
        devolver x[1]

  si n > 2
```

```
si n es impar
  medio = n / 2
sino
  medio = n / 2 - 1;
si x[medio] > y[n/2]
  devolver medianaRecursiva(x[0..medio], y[n/2..n-1]);
sino
  devolver medianaRecursiva(x[medio..n-1], y[0..n/2]);
```

Con respecto a la implementación del algoritmo, la misma resultó bastante sencilla de realizar en el lenguaje C++, utilizando las operaciones básicas que provee el lenguaje y usando arreglos básicos para la reprentación de los vectores. La única parte no trivial radica en el uso de punteros para lograr llamar recursivamente a la funcón con las "mitades" correspondientes de cada arreglo. También, cabe notar que en la implementación fue necesario pasar como parámetro extra el tamaño de los arreglos que reprentan a los vectores por cuestiones intrínsicas al manejo de arreglos dinámicos en C++.

Como se puede apreciar, el algoritmo sigue los pasos de un algoritmo clásico de *divide and conquer* dividiendo el problema en arreglos de menor tamaño al ir descartando elementos mayores y menores a la mediana. De esta manera, si logramos mostrar que en cada llamado recursivo la mediana original es la misma que la de los dos nuevos arreglos, entonces podremos asegurar que el algoritmo es correcto. Para mostrar esto primero veremos que en cada paso la mediana sigue estando dentro de alguno de los arreglos, y luego que los números menores o iguales a la mediana descartados son la misma cantidad que los números mayores o iguales a la misma que se descartan.

Demostración de que la mediana se encuentra en el próximo llamado recursivo

Dividiremos la demostración en dos casos, cuando la longitud de los arreglos iniciales es impar y cuando es par.

Sean X e Y los arreglos iniciales, entonces si llamo A a $concatenar(X, Y)$ tengo que $ordenar(A) = [a_1, a_2, .., a_n, .., a_{2n}]$, como la longitud de A es par, su mediana es a_n. Se puede ver entonces que la mediana tiene $n-1$ elementos menores o iguales a ella ($A[1..n-1]$) y n elementos mayores o iguales ($A[n+1..2n]$).

1. Longitud impar:

   Los arreglos que tomará la función para el llamado recursivo estan dados por la comparación entre las medianas de los mismos siendo $x_{(n+1)/2}$ la mediana de X e $y_{(n+1)/2}$ la mediana de Y.

   (a) $x_{(n+1)/2} \geq y_{(n+1)/2}$:
   En este caso el algoritmo toma como nuevos arreglos a $X[1..(n+1)/2]$ e $Y[(n+1)/2..n]$. Supongo ahora, que m (a partir de ahora nos referiremos asi a la mediana) no está en los nuevos arreglos, es decir, pertenece a $X((n+1)/2..n]$ o a $Y[1..(n+1)/2)$.

      i. Si $m \in X((n+1)/2..n] \implies m \geq x_{(n+1)/2}$. Si $m = x_{(n+1)/2}$ entonces SI pertenece a los nuevos arreglos pues $x_{(n+1)/2}$ pertenece. Si no son iguales entonces $m > x_{(n+1)/2} \implies m > x \ \forall x \in X[1..(n+1)/2]$. Ademas si $m > x_{(n+1)/2} \geq y_{(n+1)/2} \implies m > y \ \forall y \in Y[1..(n+1)/2]$. Como $X[1..(n+1)/2]$ y $Y[1..(n+1)/2]$ tienen $(n+1)/2$ elementos cada uno entonces m es mayor a $2((n+1)/2) = n+1$ elementos lo que es absurdo porque la mediana solo puede tener tener $n-1$ elementos menores por lo visto anteriormente.

      ii. Si $m \in Y[1..(n+1)/2) \implies m \leq y_{(n+1)/2}$. Si $m = y_{(n+1)/2}$ entonces SI pertenece a los nuevos arreglos pues $y_{(n+1)/2}$ pertenece. Si no son iguales entonces $m < y_{(n+1)/2} \implies m < y \ \forall y \in Y[(n+1)/2..n]$. Ademas si $m < y_{(n+1)/2} \leq x_{(n+1)/2} \implies m < x \ \forall x \in X[(n+1)/2..n]$. Como $X[(n+1)/2..n]$ y $Y[(n+1)/2..n]$ tienen $(n+1)/2$

elementos cada uno entonces m es menor a $2((n+1)/2) = n+1$ elementos lo que es absurdo porque la mediana solo puede tener tener n elementos mayores por lo visto anteriormente.

De esta manera queda demostrado que si n es impar y $x_{(n+1)/2} \geq y_{(n+1)/2}$ entonces la mediana pertenece a alguno de los nuevos arreglos pasados como parámetro.

(b) $x_{(n+1)/2} < y_{(n+1)/2}$:
En este caso el algoritmo toma como nuevos arreglos a $X[(n+1)/2..n]$ e $Y[1..(n+1)/2]$. Supongo ahora, que m no está en los nuevos arreglos, es decir, pertenece a $X[1..(n+1)/2)$ o a $Y((n+1)/2..n]$.

i. Si $m \in X[1..(n+1)/2) \implies m \leq x_{(n+1)/2}$. Si $m = x_{(n+1)/2}$ entonces SI pertenece a los nuevos arreglos pues $x_{(n+1)/2}$ pertenece. Si no son iguales entonces $m < x_{(n+1)/2} \implies m < x \ \forall x \in X[(n+1)/2..n]$. Ademas si $m < x_{(n+1)/2} < y_{(n+1)/2} \implies m < y \ \forall y \in Y[(n+1)/2..n]$. Como $X[(n+1)/2..n]$ y $Y[(n+1)/2..n]$ tienen $(n+1)/2$ elementos cada uno entonces m es menor a $2((n+1)/2) = n+1$ elementos lo que es absurdo porque la mediana solo puede tener tener $n-1$ elementos menores por lo visto anteriormente.

ii. Si $m \in Y((n+1)/2..n] \implies m \geq y_{(n+1)/2}$. Si $m = y_{(n+1)/2}$ entonces SI pertenece a los nuevos arreglos pues $y_{(n+1)/2}$ pertenece. Si no son iguales entonces $m > y_{(n+1)/2} \implies m > y \ \forall y \in Y[1..(n+1)/2]$. Ademas si $m > y_{(n+1)/2} > x_{(n+1)/2} \implies m > x \ \forall x \in X[1..(n+1)/2]$. Como $X[(n+1)/2..n]$ y $Y[(n+1)/2..n]$ tienen $(n+1)/2$ elementos cada uno entonces m es mayor a $2((n+1)/2) = n+1$ elementos lo que es absurdo porque la mediana solo puede tener tener n elementos mayores por lo visto anteriormente.

De esta manera queda demostrado que si n es impar y $x_{(n+1)/2} \geq y_{(n+1)/2}$ entonces la mediana pertenece a alguno de los nuevos arreglos pasados como parámetro.

2. Longitud par

Los arreglos que tomará la función para el llamado recursivo estan dados por la comparación entre $x_{n/2}$ e $y_{n/2+1}$.

(a) $x_{n/2} \geq y_{n/2+1}$: En este caso el algoritmo toma como nuevos arreglos a $X[1..n/2]$ e $Y[n/2+1..n]$. Supongo ahora, que m no está en los nuevos arreglos, es decir, pertenece a $X(n/2..n]$ o a $Y[1..n/2+1)$.

i. Si $m \in X(n/2..n] \implies m \geq x_{n/2}$. Si $m = x_{n/2}$ entonces SI pertenece a los nuevos arreglos pues $x_{n/2}$ pertenece. Si no son iguales entonces $m > x_{n/2} \implies m > x \ \forall x \in X[1..n/2]$. Ademas si $m > x_{n/2} \geq y_{n/2+1} \implies m > y \ \forall y \in Y[1..n/2+1]$. Como $X[1..n/2]$ y $Y[1..n/2+1]$ tienen $n/2$ y $n/2+1$ elementos respectivamente entonces m es mayor a $n/2+n/2+1 = n+1$ elementos lo que es absurdo porque la mediana solo puede tener tener $n-1$ elementos menores por lo visto anteriormente.

ii. Si $m \in Y[1..n/2+1) \implies m \leq y_{n/2+1}$. Si $m = y_{n/2+1}$ entonces SI pertenece a los nuevos arreglos pues $y_{n/2+1}$ pertenece. Si no son iguales entonces $m < y_{n/2+1} \implies m < y \ \forall y \in Y[n/2+1..n]$. Ademas si $m < y_{n/2+1} \leq x_{n/2} \implies m < x \ \forall x \in X[n/2..n]$. Como $X[n/2..n]$ y $Y[n/2+1..n]$ tienen $n/2+1$ y $n/2$ elementos respectivamente entonces m es menor a $n/2+1+n/2 = n+1$ elementos lo que es absurdo porque la mediana solo puede tener tener n elementos mayores por lo visto anteriormente.

De esta manera queda demostrado que si n es par y $x_{n/2} \geq y_{n/2+1}$ entonces la mediana pertenece a alguno de los nuevos arreglos pasados como parámetro.

(b) $x_{n/2} < y_{n/2+1}$: En este caso el algoritmo toma como nuevos arreglos a $X[n/2..n]$ e $Y[1..n/2+1]$. Supongo ahora, que m no está en los nuevos arreglos, es decir, pertenece a $X[1..n/2)$ o a $Y(n/2+1..n]$.

i. Si $m \in X[1..n/2) \implies m \leq x_{n/2}$. Si $m = x_{n/2}$ entonces SI pertenece a los nuevos arreglos pues $x_{n/2}$ pertenece. Si no son iguales entonces $m < x_{n/2} \implies m < x \ \forall x \in X[n/2..n]$. Ademas si $m < x_{n/2} \leq y_{n/2+1} \implies m < y \ \forall y \in Y[n/2+1..n]$. Como $X[n/2..n]$ y $Y[n/2+1..n]$ tienen $n/2+1$ y $n/2$ elementos respectivamente entonces m es menor a $n/2+1+n/2 = n+1$ elementos lo que es absurdo porque la mediana solo puede tener tener n elementos mayores por lo visto anteriormente.

ii. Si $m \in Y(n/2+1..n] \implies m \geq y_{n/2+1}$. Si $m = y_{n/2+1}$ entonces SI pertenece a los nuevos arreglos pues $y_{n/2+1}$ pertenece. Si no son iguales entonces $m > y_{n/2+1} \implies m > y \ \forall y \in Y[1..n/2+1]$. Ademas si $m > y_{n/2+1} > x_{n/2} \implies m > x \ \forall x \in X[1..n/2]$. Como $X[1..n/2]$ y $Y[1..n/2+1]$ tienen $n/2$ y $n/2+1$ elementos respectivamente entonces m es mayor a $n/2+n/2+1 = n+1$ elementos lo que es absurdo porque la mediana solo puede tener tener $n-1$ elementos menores por lo visto anteriormente.

De esta manera queda demostrado que si n es par y $x_{n/2} \geq y_{n/2+1}$ entonces la mediana pertenece a alguno de los nuevos arreglos pasados como parámetro.

En conclusión queda demostrado que siempre que se realiza un llamado recursivo, la mediana pertenece a uno de los dos arreglos pasados como parámetro.

Demostración de que se descartan la misma cantidad de elementos mayores o iguales y menores o iguales que la mediana

Sean X e Y los arreglos pasados como parámetro y n su longituud.

1. Si la longitud de los arreglos es impar, los elemenos que se descartan dependen de la comparación entre $x_{(n+1)/2}$ e $y_{(n+1)/2}$.

   (a) Si $x_{(n+1)/2} \geq y_{(n+1)/2}$ entonces se descarta $X((n+1)/2..n]$ e $Y[1..(n+1)/2)$. Como la mediana es menor o igual que $x_{(n+1)/2}$ por estar incluída y los elementos de $X((n+1)/2..n]$ son mayores o iguales que $x_{(n+1)/2}$ entonces se descartan $(n-1)/2$ elementos mayores o iguales que la mediana. Ademas como la mediana es mayor o igual que $y_{(n+1)/2}$ por estar incluída y los elementos de $Y(1..(n+1)/2]$ son menores o iguales que $y_{(n+1)/2}$ entonces se descartan $(n-1)/2$ elementos menores o iguales que la mediana. De esta manera queda demostrado que para el caso en que n es impar y $x_{(n+1)/2}$ es mayor o igual a $y_{(n+1)/2}$ se descartan la misma cantidad de elementos menores o igual que mayores o iguales que la mediana.

   (b) Si $x_{(n+1)/2} < y_{(n+1)/2}$ entonces se descarta $Y((n+1)/2..n]$ e $X[1..(n+1)/2)$. Como la mediana es mayor o igual que $x_{(n+1)/2}$ por estar incluída y los elementos de $X[1..(n+1)/2)$ son menores o iguales que $x_{(n+1)/2}$ entonces se descartan $(n-1)/2$ elementos menores o iguales que la mediana. Ademas como la mediana es menor o igual que $y_{(n+1)/2}$ por estar incluída y los elementos de $Y((n+1)/2..n]$ son mayores o iguales que $y_{(n+1)/2}$ entonces se descartan $(n-1)/2$ elementos mayores o iguales que la mediana. De esta manera queda demostrado que para el caso en que n es impar y $x_{(n+1)/2}$ es menor o igual a $y_{(n+1)/2}$ se descartan la misma cantidad de elementos menores o igual que mayores o iguales que la mediana.

2. Si la longitud de los arreglos es par, los elementos que se descartan dependen de la comparación entre $x_{n/2}$ e $y_{n/2+1}$.

   (a) Si $x_{n/2} \geq y_{n/2+1}$ entonces se descarta $X(n/2..n]$ e $Y[1..n/2+1)$. Como la mediana es menor o igual que $x_{n/2}$ por estar incluída y los elementos de $X(n/2..n]$ son mayores o iguales que $x_{n/2}$ entonces se descartan $n/2$ elementos mayores o iguales que la mediana. Ademas como la mediana es mayor o igual que $y_{n/2+1}$ por estar incluída y los elementos de $Y[1..n/2+1)$ son menores o iguales que $y_{n/2+1}$ entonces se descartan $n/2$ elementos

menores o iguales que la mediana. De esta manera queda demostrado que para el caso en que n es par y $x_{n/2}$ es mayor o igual a $y_{n/2+1}$ se descartan la misma cantidad de elementos menores o igual que mayores o iguales que la mediana.

(b) Si $x_{n/2} < y_{n/2+1}$ entonces se descarta $Y(n/2+1..n]$ e $X[1..n/2)$. Como la mediana es mayor o igual que $x_{n/2}$ por estar incluída y los elementos de $X[1..n/2)$ son menores o iguales que $x_{n/2}$ entonces se descartan $n/2-1$ elementos menores o iguales que la mediana. Ademas como la mediana es menor o igual que $y_{n/2+1}$ por estar incluída y los elementos de $Y(n/2+1..n]$ son mayores o iguales que $y_{n/2+1}$ entonces se descartan $n/2-1$ elementos mayores o iguales que la mediana. De esta manera queda demostrado que para el caso en que n es par y $x_{n/2}$ es menor a $y_{n/2+1}$ se descartan la misma cantidad de elementos menores o igual que mayores o iguales que la mediana.

Finalmente queda demostrado que en todos los casos se descartan la misma cantidad de elementos mayores o iguales y menores o iguales que la mediana.

Juntando ambas demostraciones podemos afirmar que en cada llamado recursivo la mediana original es la misma que la mediana entre la concatenación de los dos nuevos arreglos. Debido a esto, al aplicar sucesivos llamados a la función se desembocara en un caso base el cual nos devolverá el resultado buscado.

Complejidad

En la presente sección se calculará la complejidad en el peor caso del algoritmo presentado basándose en el modelo de cómputo uniforme.

La función presenta dos casos base que se dan cuando n (variable que indica la longitud de los arreglos) toma los valores 1 ó 2. En ambos casos el algoritmo sólo realiza comparaciones entre elementos y retorna un resultado, sin entrar en ningún ciclo.

Si n es mayor a 2 primero se realizan asignaciones y operaciones simples (tiempo constante) para luego efectuar un llamado recursivo. En el caso en que el tamaño de los arreglos es par, se toma $n = n/2$ como nuevo parámetro de longitud. En el caso impar, la longitud de los arreglos será $(n+1)/2$. De esta manera, cuando n es potencia de 2, el algoritmo ejecuta $\log(n)$ llamados recursivos hasta llegar a un caso base. De lo contrario se producen $\log(n)+1$ llamadas (se refiere a la parte entera de $\log(n)$).

Por lo tanto la función que determina la complejidad es:

$$T(n) = c + T(n/2) = c + c + T(n/4) = .. = \sum_{i=1}^{\log(n)+1} c = c \sum_{i=1}^{\log(n)+1} 1 = c(\log(n)+1)$$

$$\implies T(n) \in O(\log(n))$$

Por último se estudiará la complejidad en relación al tamaño de la entrada. Sea t el tamaño de la entrada, X e Y los arreglos pasados como parámetro y n su tamaño.

$$t = \log(n) + \sum_{i=1}^{n} \log(X_i) + \sum_{i=1}^{n} \log(Y_i) > \log(n) + \sum_{i=1}^{n} 1 + \sum_{i=1}^{n} 1$$

$$= \log(n) + 2n > \log(n)$$

$$\implies \text{como } T(n) \in O(\log(n)) \text{ y } \log(n) < t \implies T(t) \in O(t)$$

Análisis de resultados

Con el propósito de analizar la eficiencia del algoritmo propuesto se construyó una simple aplicación de prueba (ver pseudocódigo debajo) que calcula el tiempo de ejecucion de la función

medianaRecursiva para vectores aleatorios de tamaño variable entre 2 y $2 * 10^8$. En principio, las pruebas realizadas diferían de las mencionadas anteriormente, ya que el tamaño de las instancias probadas era mucho menor (entre 1 y 10^4). Sin embargo, como se verá más adelante, los resultados para este tipo de pruebas no reflejaba datos significativos para el análisis del algoritmo y su complejidad.

```
para i desde 2 hasta 2*10^8
vector x[1..i]
vector y[1..i]

x[1] = numero aleatorio entre 0 y 20
y[1] = numero aleatorio entre 0 y 20

para j desde 2 hasta i
x[j] = x[j-1] + numero aleatorio entre 0 y 20
y[j] = y[j-1] + numero aleatorio entre 0 y 20

T = tiempo que tarda la funcion medianaRecursiva(x,y,i) en microsegundos

guardar en Ej3-Complejidad.txt la linea "i T"
i = i*2
```

A continuación se presentan los resultados de la prueba realizada.

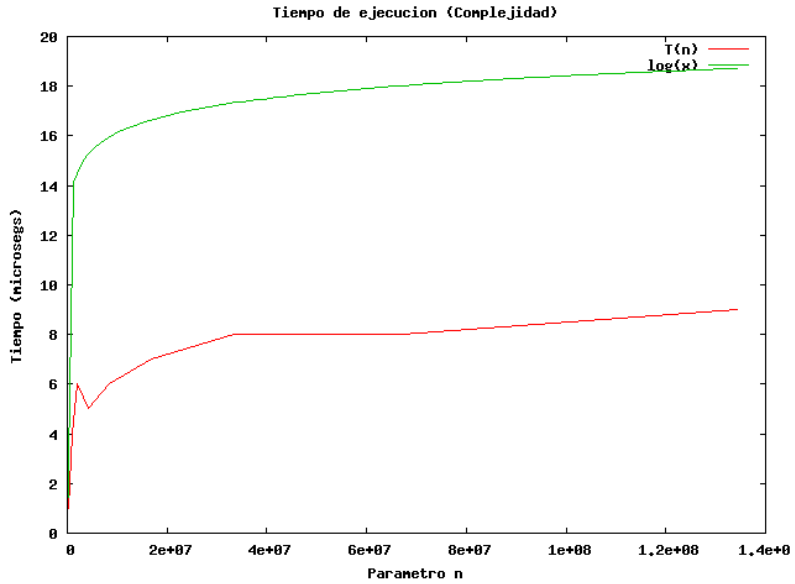

Figura 3.1

Como se puede observar en el gráfico, los tiempos de ejecución del algoritmo son muy bajos en relación al tamaño de la entrada. Esta fue la razón por la que se descartó probar sólo casos pequeños para verificar la eficiencia del algoritmo y se optó por analizar casos en el que el tamaño de la entrada creciera exponencialmente. A simple vista se puede observar que el gráfico de la función que mide el tiempo tiene el mismo comportamiento que la función de complejidad teórica. De este modo, además de la demostración teórica de la complejidad, se obtiene una prueba empírica de que esta misma es correcta.

Conclusiones

Este ejercicio nos hizo utilizar la tćnica de Divide and Conquer y nos mostró varias de sus características. La necesidad de utilizar esta técnica resultó de la búsqueda de disminuir la complejidad lineal propuesta por la solucin trivial. La complejidad obtenida muestra la relación que hay en ciertos algoritmos Divide and Conquer, entre el costo temporal de resolverlo y la cantidad de elementos de entrada. En este caso ni siquiera fue necesario leer cada uno de los elementos gracias a que contábamos con el orden de los vectores.

En nuestro algoritmo el costo de cada una de las divisiones y cada una de las conquistas es contante, permitiendo que sólo la cantidad de llamadas a problemas más pequeños defina la complejidad. Esta propiedad no logra sólo ser eficiente, sino que además, permite un cálculo sencillo de la complejidad temporal. En las pruebas empíricas realizadas, puede observarse como el tiempo que toma realizar nuestro algoritmo,es notablemente inferior al del tamaño de la entrada. Cabe destacar que nuestro algoritmo hace una cantidad similar de operaciones en todos los casos (para un mismo tamaño de la entrada), ya que para calcular la mediana siempre debe llegar al caso base. Esto ayuda a la demostración de la correctitud del algortimo y lo hace más natural. Puede observarse en el gráfico de nuestras pruebas, la ausencia de grandes picos, que muestra empíricamente que para entradas de similar tamaño, el tiempo demandado será similar .

Realizar este ejercicio también nos mostró la sencillez y la naturalidad con la que se pueden implementar los algoritmos divide and conquer mediante un método recursivo, aunque claro, esto puede cambiar dependiendo de lo complejo que sea implementar la división y la conquista.

En definitiva, se puede concluir que en ciertos casos,y en este en particular, los algoritmos Divide and Conquer son fáciles de implementar, eficientes en el peor caso, y es sencillo calcular su complejidad.