Universidad de Buenos Aires
Facultad de
Ciencias Exactas y Naturales
Departamento de Computación

Algoritmos y Estucturas de datos III

Segundo Cuatrimestre de 2011

Trabajo Práctico 1

Problema1: Campeonato de Tennis.
Problema2: Pizza entre amigos.
Problema3: Conjetura de Goldbach.

Grupo: '1'

Integrante	LU	Correo electrónico
Cammi, Martín	676/02	martincammi@gmail.com
Garbi, Sebastián	179/05	garbyseba@gmail.com
Kretschmayer, Daniel	310/99	daniak@gmail.com

Índice

1. Ejecución del TP

1.1. Lenguaje utilizado

El lenguaje utilizado para el trabajo práctico ha sido *Java*, compilando con la versión 1.5 de la Virtual Machine.

Este trabajo se acompaña con los fuentes de la solución que puede importarse en cualquier Eclipse IDE.

1.2. Como ejecutar el TP

Desde cualquier Eclipse:

- Seleccionar File ⇒ Import.
- Sseleccionar General ⇒ Existing Projects into Workspace ⇒ Next.
- Seleccionar el directorio llamado Algo3Tp1.
- Finish.

Desde la vista de **Package Explorer** bajo el paquete **src** aparecerán tres paquetes màs y dentro de cada uno de ellos los siguientes archivos de java:

- problema1

  Tennis.java

  TestTennis.java

- problema2

  Pizza.java

  TestPizza.java

- problema3

  Goldbach.java

  TestGoldbach.java

Los archivos Tennis.java, Pizza.java y Goldbach.java contienen los algoritmos solución. El resto de los archivos que comienzan con **Test** son los tests asociados.

2. Problema1: Campeonato de Tennis

2.1. Introducción

2.2. Explicación de la solución

2.3. Pseudo-código

2.4. Complejidad temporal

2.5. Tests

2.6. Gráficos

2.7. Conclusiones

3. Problema2: Pizza entre amigos

3.1. Introducción

3.2. Explicación de la solución

3.3. Pseudo-código

3.4. Complejidad temporal

3.5. Tests

3.6. Gráficos

3.7. Conclusiones

4. Problema3: Conjetura de Goldbach.

4.1. Introducción

En matemática una conjetura es una afirmación que no ha sido demostrada, pero basado en pruebas empíricas parecería ser cierta. En este contexto la Conjetura de Goldbach o Conjetura fuerte de Goldbach intenta expresar una verdad una relación entre pares y primos que no ha sido aún verificada en su totalidad, dicha conjetura enuncia que:

Todo número par mayor a dos puede ser escrito como suma de dos números primos.

En el presente trabajo implentaremos un algoritmo que, basado en la Conjetura de Goldbach y dado un número par mayor a dos, retorne dos primos que sumados obtengan dicho número.

Utilizaremos la técnica de backtraking para generar posibles soluciones e ir al mismo tiempo realizando podas para no recorrer las que no los sean.

Se analizará también la complejidad del algoritmo en base al tamaño de la entrada utilizando modelos de complejidad.

Finalmente se realizarán gráficas y analizarán peores casos, casos promedios y posibles casos patológicos.

4.2. Explicación de la solución

Como precondición para el algoritmo el número n ingresado deberá ser par. A continuación el algoritmo irá generando pares de números que sumen n y que sean candidatos a solución del siguiente modo:

1 + (n-1), 2 + (n-2), ... (n/2) + (n/2), ... (n-2) + 2, (n-1) + 1

Poda1: Lo primero que notamos es que al llegar a la mitad de pares generados la siguiente mitad será similar a la anterior ya que serán los mismos sumandos pero invertidos con respecto a la suma. Entonces con verificar que sumandos son primos en la primera mitad basta para también cubir los casos de la segunda mitad.

La primera poda entonces es solo recorrer la primer mitad:

1 + (n-1), 2 + (n-2), ... (n/2) + (n/2)

Poda2: También puede verse que el número 1 no es de por si primo así que cualquier suma en que participe no será una solución válida. Podamos entonces la solución del 1.

2 + (n-2), ... (n/2) + (n/2)

Poda3: Del mismo modo, también podemos observar que cualquier número par, salvo el 2 no será primo ya que al ser par será divisible por 2. Excluimos entonces todas las sumas con pares en ellas salvo la que contenga al 2.

2 + (n-2), 3 + (n-3) ... 2m+1 (n-(2m+1)) ... (n/2)-1 + (n/2)+1

Poda4: El caso del 2 podríamos tratarlo a parte ya que la única suma en la que participa es en la del 4 donde $4 = 2 + 2$. No existe otra suma en la que el primo 2 aparezca ya que cualquier par que tenga como un sumando al 2 deberia tener como segundo sumando a otro número par y además primo y el único que cumple esa condición es el 2 caso que acabamos de excluir.

3 + (n-3) ... 2m+1 (n-(2m+1)) ... (n/2)-1 + (n/2)+1

De esta forma generamos pares de números que sumados dan el par inicial y que han sido previamente filtrados por las podas mencionadas. A continuación se verificará si alguno de ellos es solución. La definición de solución en este problema es para cada par si ambos números son primos.

Para el cálculo de si un número es primo verificaremos si el único divisor positivo además del uno es si mismo, para ello se dividirá por todos los menores.

Poda5: Un optimización que se ha tenido en cuenta es verificar los divisores solo hasta la raiz cuadrada.

Poda6: También, si al recorrer un par de sumandos notamos que ambos son iguales, verificaremos si es primo sólo uno de ellos, ya que es análogo hacerlo con el par idéntico. ej: Si el n considerado es 10 y se está evaluando el par 5 + 5, solo verificaremos si 5 es primo una sola vez.

Cabe aclarar que para optimizar el algoritmo la generación de candidatos y la verificación de solución se realizarán simultáneamente. No generaremos todas las soluciones y luego las verificaremos una a una, sino que verificaremos una a una a medida que las vayamos generando.

4.3. Pseudo-código

para cada n1,n2 candidatos podados **hacer**
  si esPrimo(n1) y esPrimo(n2) **entonces**
    retornar n1, n2
  fin si
fin para

4.4. Modelo elegido

Para el cálculo del orden utilizaremos primeramente el modelo *Uniforme* ya que las operaciones básicas que posee el algoritmo no son significativas ni parecen influir tanto como las iteraciones principales.

De todos modos realizaremos también un análisis con el modelo *logarítmico* para observar que si alguna operación básica en particular prepondera sobre otras o puede afectar el algoritmo para alguna entrada en particular

Analizaremos también los peores casos y compararemos ambas gráficas con ambos modelos.

4.5. Complejidad temporal

La siguiente complejidad se calcula en base al modelo *Uniforme*.

A continuación se describe el algoritmo donde se ha asignado a cada paso relevante las complejidades correpondientes:

```
Paso1:    Para cada n1,n2 candidatos O(n/4)
Paso2:       Si esPrimo(n1) y esPrimo(n2) entonces  O(\sqrt{n1}) + O(\sqrt{n2})
Paso3:         retornar n1, n2
Paso4:       Fin Si
Paso5:    Fin Para
```

Paso1: Comenzaremos recorriendo de todos los posibles candidatos solo la mitad, ya que la otra mitad corresponde a soluciones análogas (Poda1) recorreremos entonces n/2 pares de cantidatos. De esa mitad solo interesarán los pares pares ya que si alguno no lo fuese sería primo (el caso de 4 se tratará aparte) de este modo de los n/2 pares recorreremos solo la mitad es decir n/4. La complejidad de este ciclo es de O(n/4).

Paso2: Este paso basará su complejidad en la complejidad de *esPrimo*. esPrimo dado un entero positivo n recorre todos los valores menores verificando si alguno es divisor. Una optimización ha sido recorrer hasta valores hasta la raiz de este modo la complejidad de esPrimo es de $O(\sqrt{n})$ que de todos modos diremos que es acotable por O(n).

De esta forma calculamos el costo del algoritmo recorriendo solo los pares impares hasta la mitad. Para ayudar a calcular este orden definamos entonces la función verificarSolución(i,n) como esPrimo(i) $\wedge$ esPrimo(n-i);

$$\sum_{i=1}^{n/4} verificarSolucion(2 \cdot i + 1, n).$$

$$\sum_{i=1}^{n/4} esPrimo(2 \cdot i + 1) + esPrimo(n - (2 \cdot i + 1)).$$

Como esPrimo(n) es O($\sqrt{(}n)) \subset$ O(n) es acotable por O(n).

$$\sum_{i=1}^{n/4} (2 \cdot i + 1) + n - (2 \cdot i + 1).$$

Simplificando los términos $(2 \cdot i + 1)$:

$$\sum_{i=1}^{n/4} n$$

$\frac{n}{4} \cdot n = \frac{n^2}{4}$ es $O(n^2)$.

Ahora bien, para describir la complejidad en función del tamaño de la entrada debemos tener en cuenta la representación del número de entrada n en la computadora. Un número n será representado como $\log n$ bits.

Definimos entonces el tamaño de entrada como:

$$T = \log n$$

Si respejamos n para describir la complejidad en función de T:

$$2^T = n$$

Como la complejidad de nuestro algoritmo es de $O(n^2)$ aplicamos cuadrado a ambos lados para ver la complejidad en T:

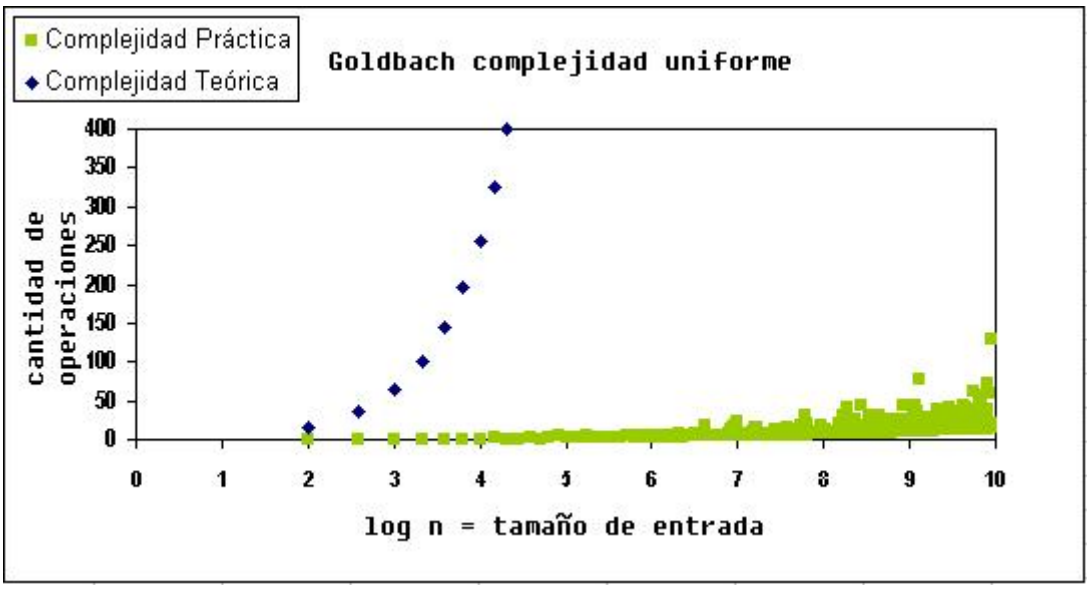
$$(2^T)^2 = n^2$$

$$(2^{2T}) = n^2$$

La complejidad temporal en base a T es exponencial del orden de $O(2^{2T})$

4.6. Tests

4.7. Gráficos



4.8. Conclusiones