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: Torneo.
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

0.1. Lenguaje utilizado . . . 3
0.2. Como ejecutar el TP . . . 3

1. Problema1: Torneo **5**
1.1. Introducción . . . 5
1.2. Explicación de la solución . . . 5
1.3. Pseudo-código . . . 6
1.4. Modelo Elegido . . . 6
1.5. Complejidad . . . 7
1.6. Tests . . . 9
1.7. Gráficos . . . 9
1.8. Conclusiones . . . 12

2. Problema2: Pizza entre amigos **13**
2.1. Introducción . . . 13
2.2. Explicación de la solución . . . 13
2.3. Pseudo-código . . . 15
2.4. Modelo Elegido . . . 16
2.5. Complejidad . . . 16
2.6. Tests . . . 17
2.7. Gráficos . . . 19
2.8. Conclusiones . . . 21

3. Problema3: Conjetura de Goldbach **23**
3.1. Introducción . . . 23
3.2. Explicación de la solución . . . 23
3.3. Pseudo-código . . . 25
3.4. Modelo Elegido . . . 25
3.5. Complejidad temporal . . . 26
3.6. Tests . . . 28
3.7. Gráficos . . . 30
3.8. Conclusiones . . . 34

Ejecución del TP

0.1. Lenguaje utilizado

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

El trabajo se acompaña con los fuentes de la solución que puede importarse en IDE de Eclipse o ejecutarse desde línea de comandos.

0.2. Como ejecutar el TP

Desde línea de comandos

- Posicionarse en el directorio Algo3Tp1
- Copiar allí el archivo de entrada para el problema i, por ejemplo Ej1.in
- Ejecutar el comando: *java -cp ./bin problema1.Ej1*

Esto generará el archivo Ej1.out con la solución en el mismo directorio Algo3Tp1.

Desde el Eclipse

Primero importaremos el proyecto:

- Seleccionar File ⇒ Import.
- Seleccionar 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 más y dentro de cada uno de ellos los siguientes archivos de java:

Para ejecutar un problema:

- Posicionarse en el directorio Algo3Tp1
- Copiar en el directorio Algo3Tp1 el archivo de entrada para el problema i, por ejemplo Ej1.in
- Con botón derecho Run As ⇒ Java Application. Se ejecutará el problema seleccionado.

Esto generará el archivo Ej1.out con la solución en el mismo directorio Algo3Tp1.

1. Problema1: Torneo

1.1. Introducción

Debemos organizar un torneo que involucre a n competidores. Cada competidor debe jugar exactamente una vez con cada uno de sus oponentes. Además, cada competidor debe jugar un partido por día con la sola posible excepción de un día en el cual no juegue.

Para ello, implentaremos un algoritmo que utilice la técnica de divide y vencerás. Si la cantidad de competidores es potencia de 2 o es par, el torneo debe terminar en $n - 1$ días. En cambio si n es impar, el torneo debe terminar en n días.

1.2. Explicación de la solución

Para que un torneo se pueda armar, el número de competidores debe ser mayor o igual a 2. En cambio, en caso de venir menor cantidad de jugadores, el algoritmo no creará el torneo, ya que no tiene sentido realizarlo.

Para resolver el problema, utilizamos una matriz como estructura de almacenamiento, donde iremos guardando la tabla de los partidos entre cada jugador, y en que dias éstos competirán. La matriz tiene como filas, la cantidad de jugadores del torneo, y como columnas la cantidad de dias que demandará este. En ambos casos, no utilizamos el índice 0, solo para facilitar la legibilidad del algoritmo.

En cada llamada recursiva, el torneo se divide de a mitades, pero nosotros llamamos a recursión solo con la primera de ellas, y luego realizamos la etapa de combinación, que termina calculando las competiciones para la otra mitad de jugadores.

Cuando la cantidad de competidores totales (o en cualquier llamada recursiva) es impar, lo que realiza el algoritmo es la de agregar un jugador «ficticio», y se genera el torneo como si este fuese uno real. Para distinguirlo, utilizamos una función que pone a ese jugador como el jugador 0. Luego, es la etapa de combinación la encargada de asignarle a ese jugador su oponente para ese día, o caso contrario, implicará que el jugador tiene fecha libre ese día, por lo que la matriz final quedará en 0.

El caso base del algoritmo es cuando quedan 2 jugadores, que los hace competir en el día 1 entre si.

1.3. Pseudo-código

Algorithm 1 Torneo

```
crearTorneo(cantJugadores)
if cantJugadores == 2 then
  fixturePartidos[1][1] = 2;
  fixturePartidos [2][1] = 1;
else
  if (cantJugadores mod 2)!=0 then
    //cantidad jugadores impar
    torneo(cantJugadores+1);
    colocarJugadorFicticio(cantJugadores);
  else
    //si cantJugadores es par
    mitadJugadores = cantJugadores / 2;
    torneo(mitadJugadores); // primero el cuadrante sup. izq.
    boolean esPar = (mitadJugadores mod 2) ==0;
    cuadranteInferiorIzquierdo(mitadJugadores, cantJugadores, esPar);
    cuadranteInferiorDerecho(mitadJugadores, cantJugadores, esPar);
    cuadranteSuperiorDerecho(mitadJugadores, cantJugadores, esPar);
  end if
end if
```

1.4. Modelo Elegido

Para calcular la complejidad de este algoritmo utilizaremos el modelo $Uniforme$, ya que la cantidad de jugadores de un torneo no es significativamente grande, por lo que la cantidad de bits para representar cualquier jugador no debería ser grande, por lo que consideramos que las operaciones para sumar y restar es constante y podemos asumirla en $O(1)$

1.5. Complejidad

```
cuadranteInferiorIzquierdo(mitadJugadores, cantJugadores, esPar)
if esPar then
  for cada jugador entre mitadJugadores+1 y cantJugadores do
    for cada dia entre 1 y mitadJugadores-1 do
      fixturePartidos[jugador][dia] = fixturePartidos[jugador-mitadJugadores][dia] + mitad-
      Jugadores;
    end for
  end for
else
  for cada jugador entre mitadJugadores + 1 y cantJugadores do
    for cada dia entre 1 y mitadJugadores do
      if fixturePartidos[jugador - mitadJugadores][dia] == 0 then
        fixturePartidos[jugador][dia]=0;
      else
        fixturePartidos[jugador][dia]= fixturePartidos[jugador-mitadJugadores][dia] + mitad-
        Jugadores;
      end if
    end for
  end for
  sacarJugadorFicticio(mitadJugadores);
end if
colocarJugadorFicticio(cantJugadores)
for cada jugador entre 1 y cantJugadores do
  for cada dia entre 1 y cantJugadores do
    if fixturePartidos[jugador][dia] == cantJugadores+1 then
      fixturePartidos[jugador][dia]= 0;
    end if
  end for
end for
sacarJugadorFicticio(mitadJugadores)
for cada jugador entre 1 y mitadJugadores do
  for cada dia entre 1 y mitadJugadores do
    if fixturePartidos[jugador][dia] == 0 then
      fixturePartidos[jugador][dia] = jugador + mitadJugadores;
      fixturePartidos[jugador+mitadJugadores][dia] = jugador;
    end if
  end for
end for
```

El costo del algoritmo es el orden que tarda la combinación más el costo de dividir.

Si analizamos la combinación, veamos primero cual es el orden de cuadranteInferiorIzquierdo. Este recorre $n/2$ jugadores, y para cada uno de ellos, chequea $n/2$ dias para realizar los cruces en el torneo. El tiempo que se demora para hacer esta iteración es de $O((n/2)^2$. Además, luego de crear los partidos entre estos competidores, si fue llamado con un número impar, debe sacar los jugadores ficticios, que nuevamente recoore n/2 jugadores, $n/2$ días. Por lo que, en el peor caso (si n es impar), el algoritmo tiene costo $O(n^2/2)$.

El análisis para cuadranteInferiorDerecho, y cuadranteSuperiorDerecho es anáalogo el de cuadranteInferiorIzquierdo, pero no debe colocar al jugador ficticio, por lo que el orden de cada uno es de $O((n/2)^2)$.

Por lo que, si sumamos los 3 cuadrantes, más colocar el jugador ficticio, el tiempo que se demora es $O(n^2)$. Notesé también, cuando la llamada recursiva tiene cantidad de jugadores impar, debe sacar al jugador ficticio que se agrego previamente, y el orden de este algoritmo también es de $O(n^2)$.

Como realizamos siempre la subdivisión en 2 veces, la cantidad de subdivisiones que realiza el algoritmo es $log(n)$ veces.

Por lo tanto, el algoritmo se demora en total, $O(n^2 * log(n))$.

Ahora debemos calcular el algoritmo en función del tamaño de entrada.

$$T = \log n$$

ya que para representar un número n, se necesitan $\log n$ bits, entonces

$$n = 2^T.$$

La complejidad del algoritmo en función del tamaño de entrada es entonces de

$$O(2^T \cdot log 2^T)$$

Que equivale a:

$$O(2^T \cdot T)$$

1.6. Tests

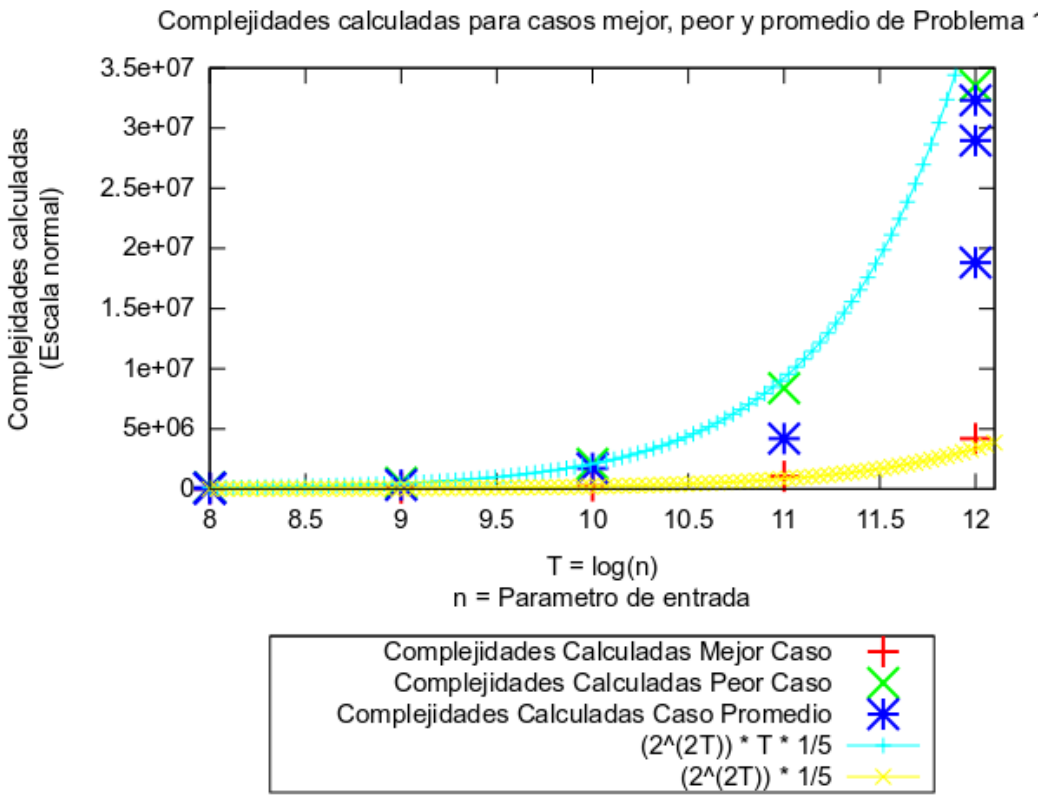
Si estudiamos previamente como está formado el algoritmo, podemos preveer que este se comportará mejor con números que son potencia de 2, ya que no debe colocar ni sacar jugadores ficticios. Por otro lado, con que más veces realice esta operación de agregar y sacar jugadores que no forman parte del torneo, el algoritmo a priori, debería ser más costoso. Realizaremos distintas mediciones, para conocer si estas previsiones son reales o no. Trataremos de realizar torneos con 3 tipos de datos de entrada.

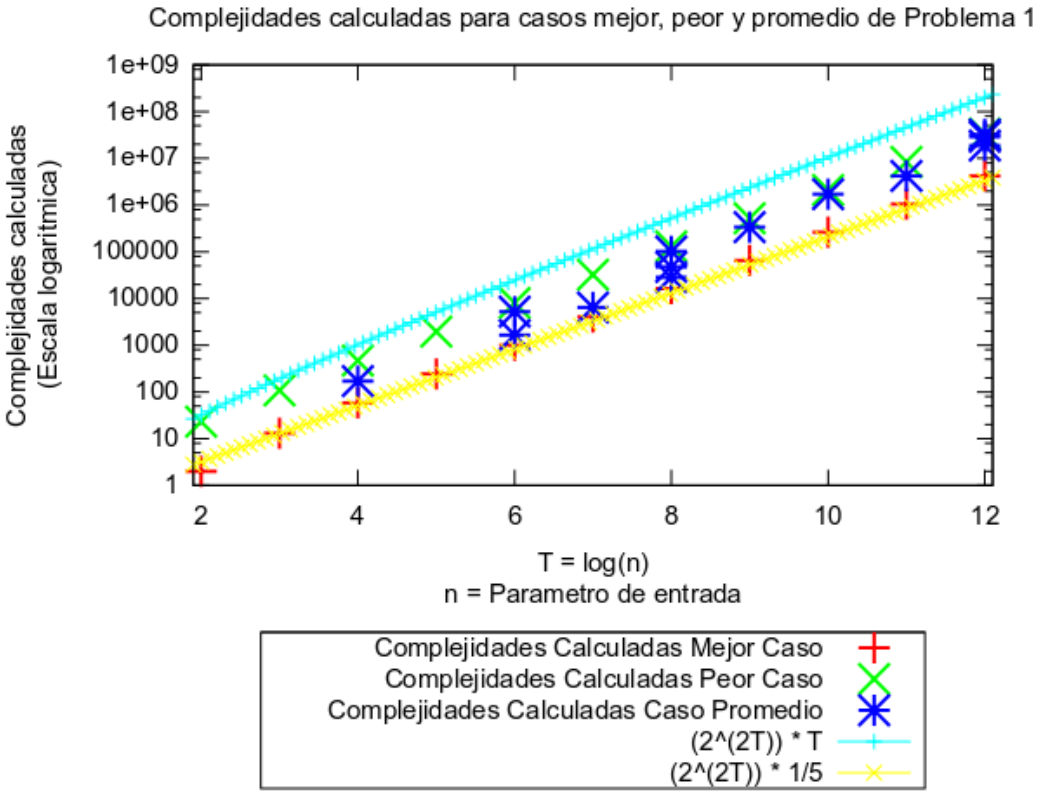
Tipo 1) Usaremos números potencia de 2, para ver si efectivamente, el algoritmo presenta un mejor caso para este tipo de torneos.

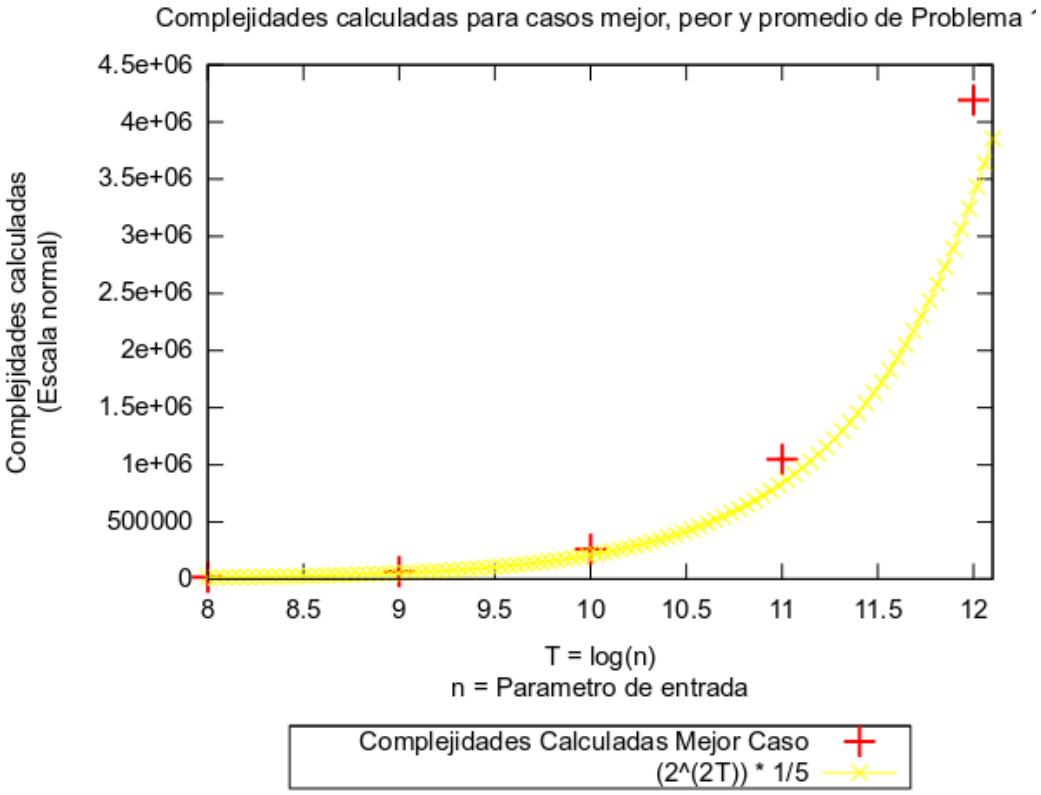
Tipo 2) Realizaremos torneos con cantidad de jugadores impares, donde este nḿero sea igual a un nḿero potencia de 2 sumado 1. Con este tipo de valores (y más para números grandes) trataremos de corroborar si efectivamente es un peor caso o no.

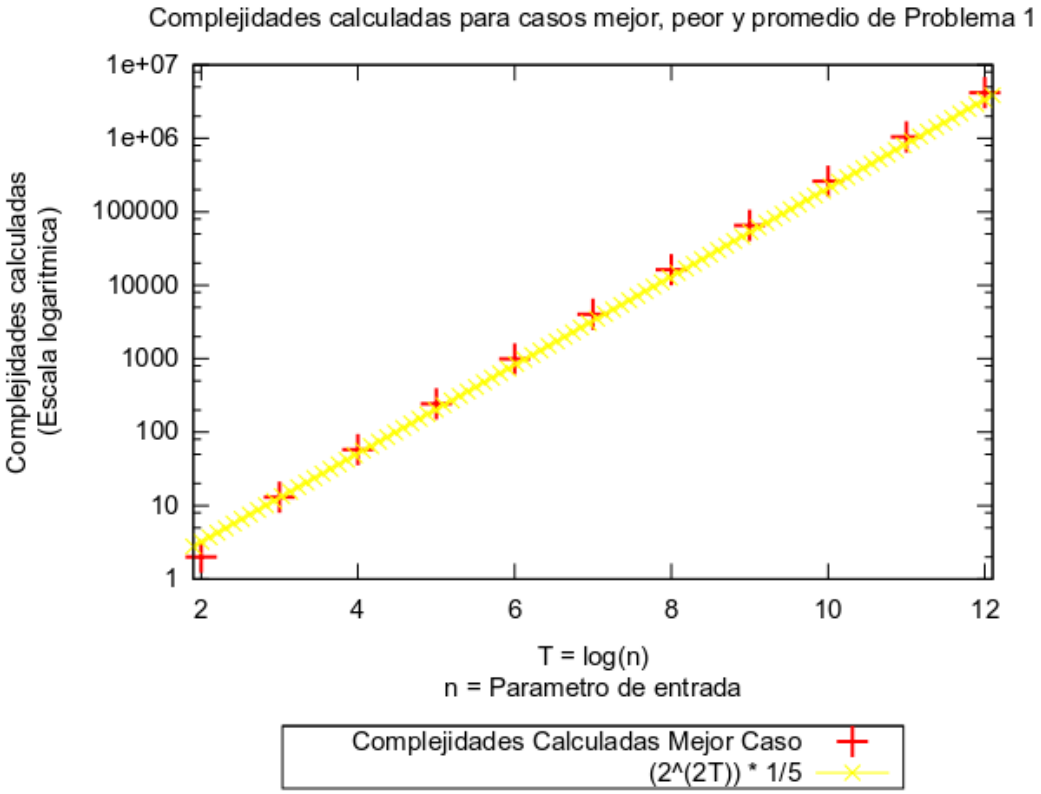
Tipo 3) Por último, haremos mediciones con números que no pertenezcan a los primeros 2 tipos, para ver como se comporta el algoritmo, y si estamos en presencia de algún mejor o peor caso.

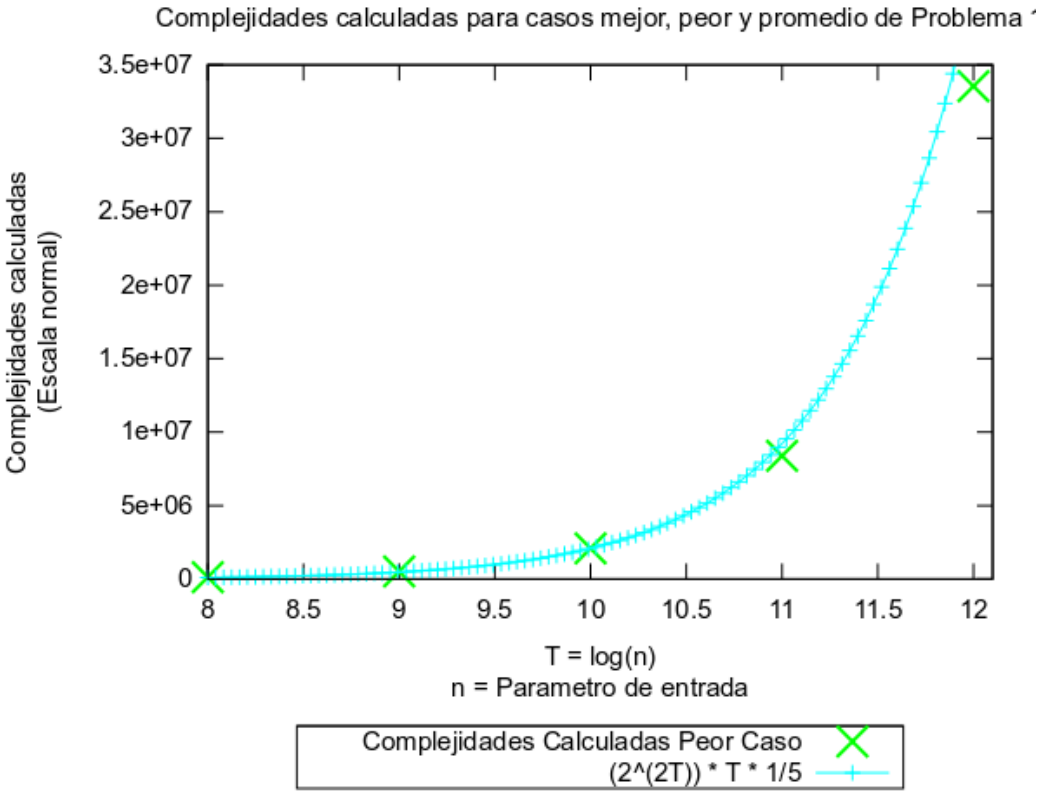
1.7. Gráficos

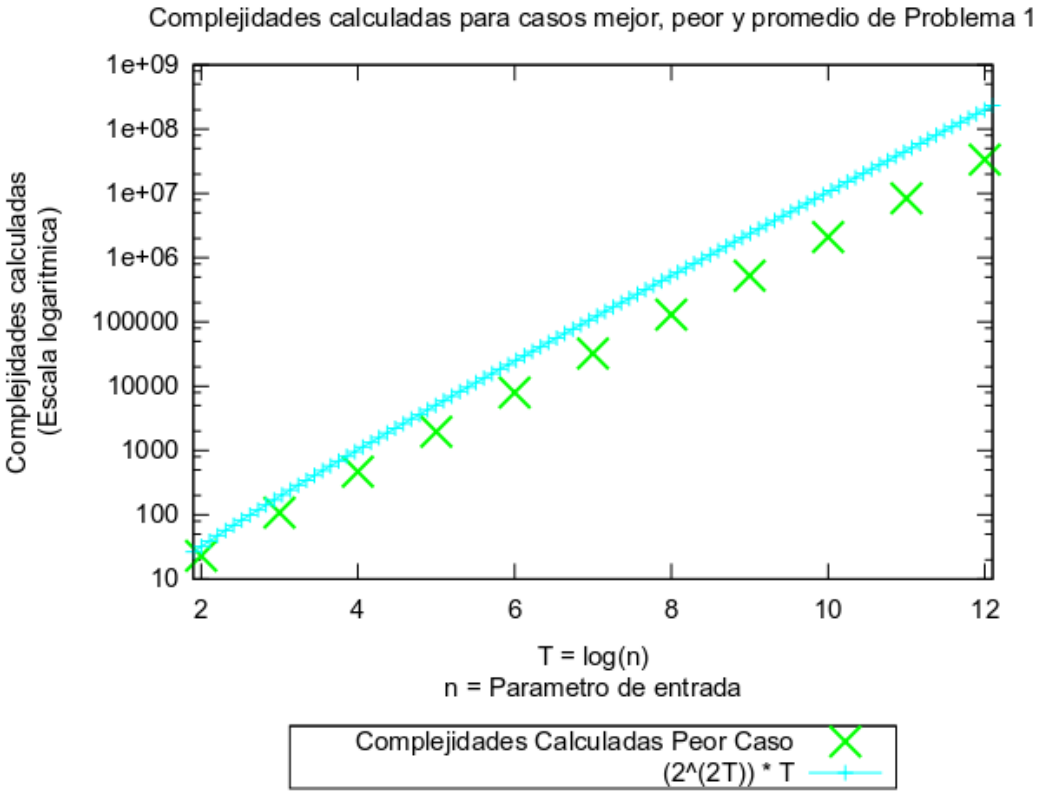



Complejidades calculadas para casos mejor, peor y promedio de Problema 1
Complejidades calculadas
(Escala logaritmica)
1e+09
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
2
4
6
8
10
12
T = log(n)
n = Parametro de entrada
Complejidades Calculadas Mejor Caso
Complejidades Calculadas Peor Caso
Complejidades Calculadas Caso Promedio
(2^(2T)) * T
(2^(2T)) * 1/5


Complejidades calculadas para casos mejor, peor y promedio de Problema
Complejidades calculadas
(Escala normal)
4.5e+06
4e+06
3.5e+06
3e+06
2.5e+06
2e+06
1.5e+06
1e+06
500000
0
8
8.5
9
9.5
10
10.5
11
11.5
12
T = log(n)
n = Parametro de entrada
Complejidades Calculadas Mejor Caso
(2^(2T)) * 1/5


Complejidades calculadas para casos mejor, peor y promedio de Problema 1
Complejidades calculadas (Escala logaritmica)
1e+07
1e+06
100000
10000
1000
100
10
1
2
4
6
8
10
12
T = log(n)
n = Parametro de entrada
Complejidades Calculadas Mejor Caso
(2^(2T)) * 1/5


Complejidades calculadas para casos mejor, peor y promedio de Problema
Complejidades calculadas (Escala normal)
3.5e+07
3e+07
2.5e+07
2e+07
1.5e+07
1e+07
5e+06
0
8
8.5
9
9.5
10
10.5
11
11.5
12
T = log(n)
n = Parametro de entrada
Complejidades Calculadas Peor Caso
(2^(2T)) * T * 1/5



1.8. Conclusiones

Hemos podido notar, que el algoritmo se comporta de acuerdo a lo esperado. Los mejores casos del algoritmo es cuando la cantidad de jugadores es potencia de 2, y los peores son cuando la cantidad de jugadores es de la forma $(2^i + 1)$, con i cualquier número entero mayor a 1. Por otro lado, los números restantes se comportan de acuerdo a la cantidad de veces que deban realizar estas operaciones de agregar y sacar ficticios, pero no son ni los mejores, ni los peores casos.

2. Problema2: Pizza entre amigos

2.1. Introducción

Se quiere pedir una pizza extra gigante para un grupo de amigos. Cada uno de los amigos tiene su preferencia de elementos que quiere que estén en la pizza y elementos que preferiría que no estén. Como quizás no sea posible satisfacer todas las preferencias de cada uno de los amigos, se pretende buscar una solución donde al menos una de las preferencias de cada amigo sea satisfecha ó, de no encontrarse una solución, responder que no es posible.

2.2. Explicación de la solución

Antes de explicar la solución elegida, comentaremos algunas decisiones tomadas ad hoc:

- En la entrada puede llegar a venir en las preferencias de alguno de los comensales un ingrediente repetido. Se descartaran las apariciones repetidas de ese ingrediete en caso que la decisión tomada sobre el mismo sea la misma. Por ejemplo las instancia +A+B+A será equivalente a la instancia +A+B. Si alguna de las repeticiones tiene la decisión contraria por ejemplo en el caso de +A+B-A, entonces asumiremos que este comensal está satisfecho ya que cualquier decisión tomada sobre el ingrediente en cuestión lo satisfacerá.
- Otro caso ad hoc es cuando en la entrada viene una linea vacia (solo ';'), en este caso asumiremos que no existe una pizza que satisfaga al menos una preferencia de este comensal ya que este no tiene preferencias.

Fuera de estos casos, asumimos que todos los comensales tienen al menos una preferencia para la pizza.

La idea del algoritmo es, usando backtracking, ir recorriendo un arbol de decisión donde la altura esta dada por la cantidad de ingredientes posibles + 1 y cada nivel i del arbol se corresponde al iesimo ingrediente cuyos subarboles son de las decisiones de poner o no ese ingrediente en la Pizza.

En las hojas de este árbol estan todas las posibles combinaciones de pizza, si ya se revisaron todas las posibles combinaciones y ninguna cumplió entonces no hay solución para esas preferencias.

En cada iteración del algoritmo se van marcando los comensales que fueron satisfechos con la desición tomada sobre el ingrediente actual, cuando se llega al último ingrediente si todos los comensales estan marcados se devuelve esa pizza, si no se desmarcan y se prueba por otra rama del arbol y se repite el proceso.

Poda1: Se puede ir chequeando a medida que se va armando la solución parcial si ya todos los comensales fueron marcados, si pasó esto se puede terminar antes ya que la solución parcial

es la misma que no poner el resto de los ingredientes.

Poda2: En cada iteración se chequean solo los comensales que no hayan sido marcados ya en la solución parcial actual, ya que los demas están satisfechos y seguirán estándolo cualquiera sea la desición que se tome sobre el resto de la pizza.

Poda3: Antes de armar el árbol se revisan las preferencias de cada uno de los comensales para ver cuales son los ingredientes que realmente importan ya que si nadie tiene preferencia (ya sea por que vaya o no en la pizza) sobre algún ingrediente obviaremos revisarlo.

Poda4: Como estamos recorriendo el árbol de decisión desde la primera hasta la última letra (ingrediente) en orden, podemos saber cuando un comensal no va a poder ser satisfecho con la solución parcial actual comparando el ingrediente actual con el "Mayor.^{en} los que tiene una preferencia el comensal.

Poda5: En el caso de estar evaluando la rama true de un ingrediente y no sacar ningún comensal, esto quiere decir que de los que quedan, nadie tiene decisión sobre ese ingrediente o bien los que la tienen prefieren que no vaya. Si continúo evaluando esta rama y llego a una solución, también llegaré no poniendolo. En cambio si no llego a una solución y nadie tenía preferencia que no vaya, tampoco lo haré en la rama false. En cuyo caso me basta con evaluar sólo la rama false y me evito hacer dos veces el subarbol que queda con los proximos ingredientes.

Tip: Vamos marcando junto con los comensales que fueron satisfechos que ingrediente los satisfiso para que, si hay que retroceder en el árbol, se desmarquen solo esos y se continúe probando con otra solución.

2.3. Pseudo-código

Algorithm 2 EncontrarPizza

```
i ← PrimerIngredienteImportante
pizza ← pizzaVacia
ponerEnLaPizza ← true
if backtrack(i, ponerEnLaPizza) then
  return pizza
else
  if backtrack(i, !ponerEnLaPizza) then
    return pizza
  else
    return sin solución
  end if
end if
```

```
backtrack(i, ponerEnLaPizza)
if ponerEnLaPizza == true then
  poner i en la pizza
end if
marcar comensales satisfechos
if no hay mas comensales insatisfechos then
  return true
else
  if quedan ingredientes sin mirar then
    i ← siguienteIngrediente
    if backtrack(i, ponerEnLaPizza) then
      return true
    else
      if backtrack(i,true) then
        return true
      else
        i ← anteriorIngrediente
      end if
    end if
  end if
  desmarcar comensales satisfechos
  if ponerEnLaPizza == true then
    sacar i de la pizza
  end if
```

return false
end if

2.4. Modelo Elegido

Para calcular la complejidad de este algoritmo utilizaremos el modelo $Uniforme$, ya que por más que la cantidad de datos de la entrada crezca, las operaciones sobre cada uno de ellos es constante y podemos asumirla en $O(1)$.

2.5. Complejidad

Siendo:
n = Cantidad de Ingredientes
m = Cantidad de Comezales

Las siguientes acciones toman: $O(1)$ Preguntar si un comensal prefiere un ingrediente
$O(1)$ Preguntar si quedan comensales insatisfechos
$O(n)$ Armar la solución
$O(1)$ Mover un comensal de una lista a otra
(ya sea vaciando una lista de principio a fin o removiendo del iterador)

En peor caso el algoritmo recorre todos los nodos del arbol y estos son $2^{n+1} - 1$ o sea $O(2^n)$. Por cada vez que recorre un nodo itera sobre todos los comensales insatisfechos restantes En peor caso esto es $O(m)$.

Esto deja el algoritmo en una complejidad teorica de $O(2^n * m)$.
Considerando que n es acotado por la cantidad de letras en el alfabeto ingles (26) el algoritmo queda dependiendo solo de m o sea $O(m)$.

Para describir la complejidad en función el tamaño de entrada debemos considerar que cada comensal puede llegar a tener hasta 26 preferencias, estos son 52 caracteres por comensal

Definimos el tamaño de entrada como:

$$T = log(m * n * 2)$$

Sabiendo que n es acotado tomamos:

$$T = log(m)$$

Como la complejidad del algoritmo es $O(m)$ despejamos n en funcion de T:

$$m = 2^T$$

2.6. Tests

Tipo1: Un estudio poco profundo nos diría que los peores casos son aquellos que tiene todas las permutaciones. A estos los llamaremos de $Tipo1$
Ej.

```
3
+A+B+C;
+A+B-C;
+A-B+C;
+A-B-C;
-A+B+C;
-A+B-C;
-A-B+C;
-A-B-C;
.
```

Tipo2: Investigando un poco más el algoritmo pudimos observar que los peores casos para cada tamaño de entrada ocurrirán cuando tenga que evaluar cada ingrediente en la mayoria de los comensales.

Sabiendo que el algoritmo primero intenta poner el ingrediente y luego, en caso que no haya solución prueba no poniendolo lo obligamos a hacer el mayor número de operaciones por la rama true.

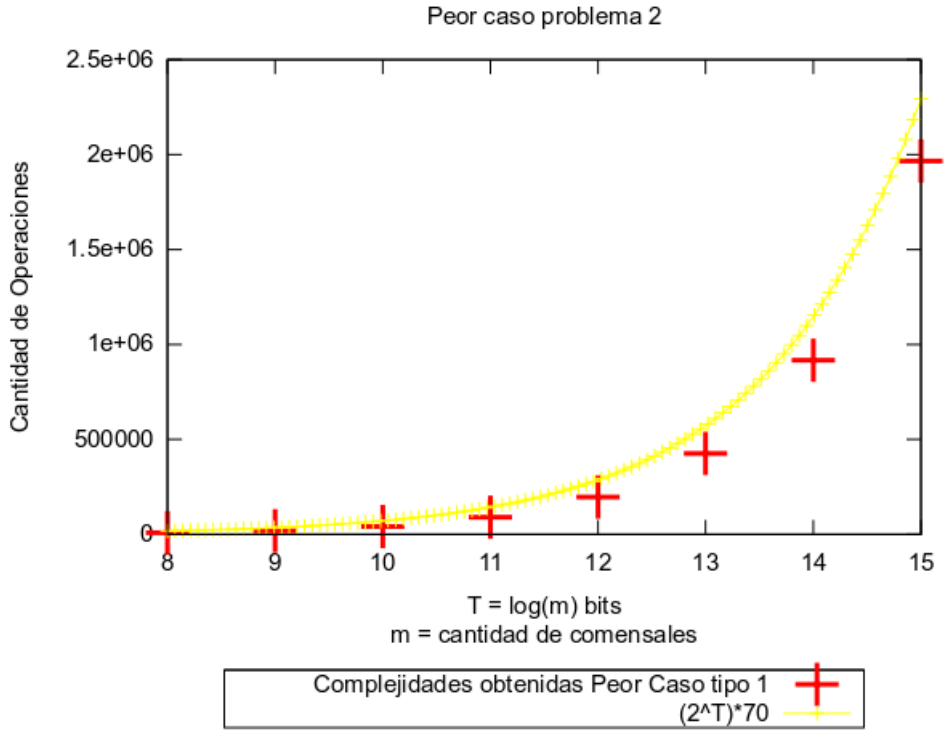
Tambien tenemos que procurar que no termine hasta haber revisado el último ingrediente así que ponemos algun comensal que tenga una preferencia contraria. Si ese comensal tiene más de una preferencia entonces se podría satisfacer evitando que se revisen más ingredientes así que no le ponemos más. Los peores casos serían de la forma:

```
3
+A+B+C;
+A+B+C;
...
+A+B-C;
+A-B;
-A;
.
```

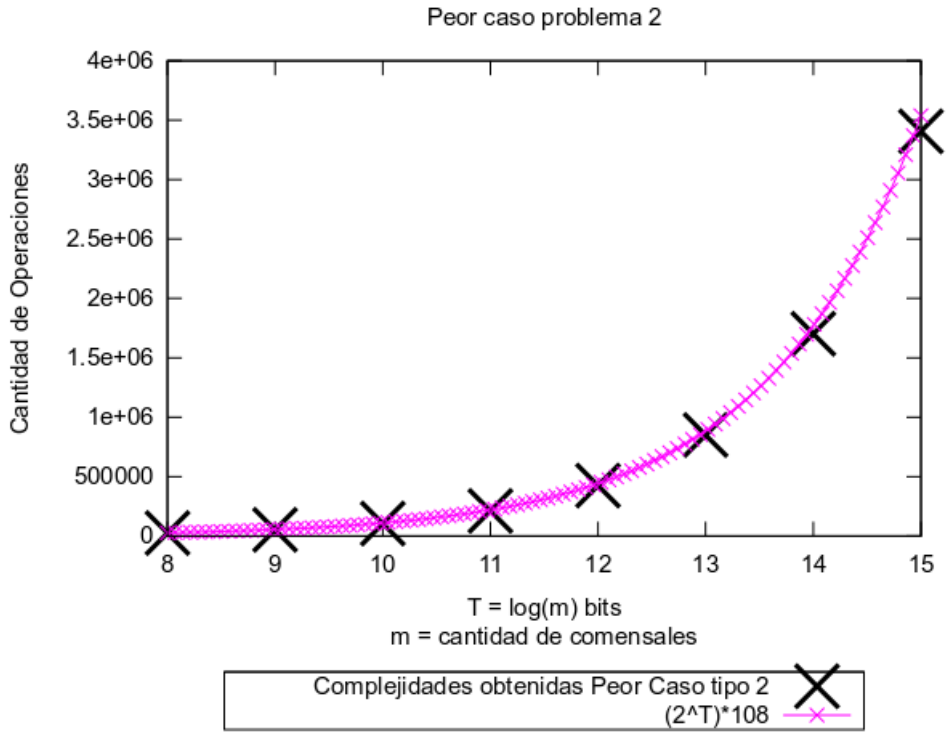
Tipo3: Finalmente observamos que los casos de *Tipo*2 descartaban la decisión de poner un ingrediente casi de inmediato así que intentamos buscando un caso peor aún y nos pusimos la meta de no descartar letras con el podado y ademas que evalúe la mayor cantidad posible de comensales en cada nodo. La solución fue la siguiente:

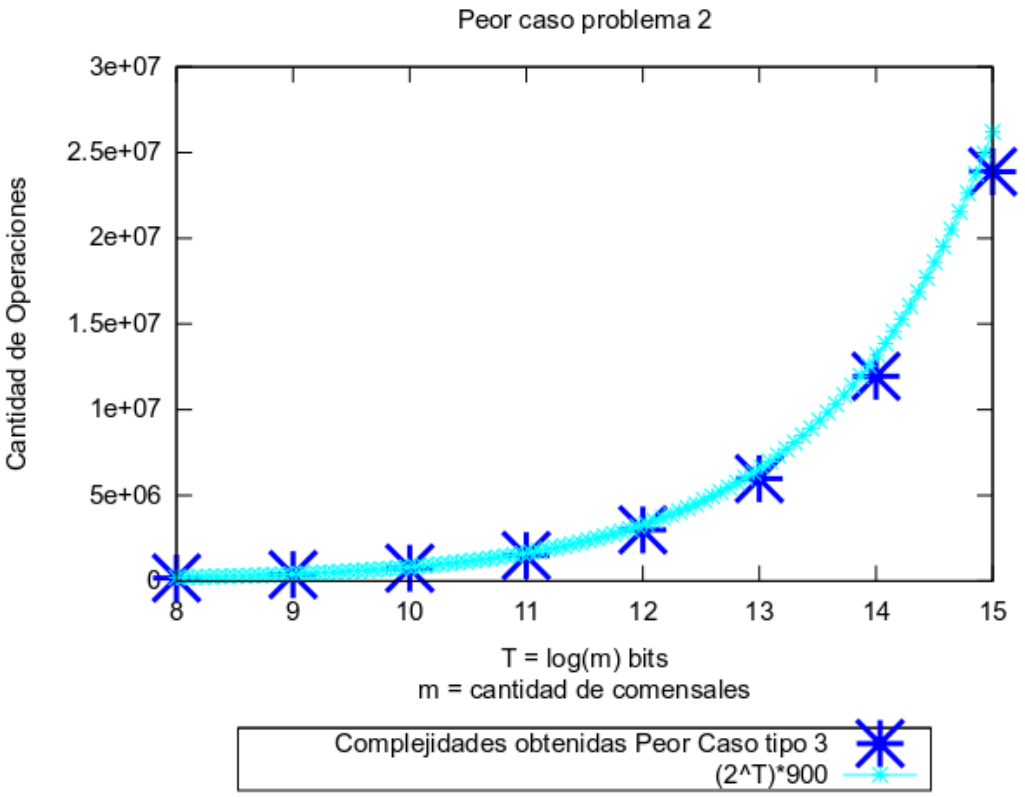
```
26
+A+B+C+D+E+F+G+H+I+J+K+L+M+N+O+P+Q+R+S+T+U+V+W+X+Y+Z;
+Z;
+Z;
 ...
+Z;
-Z;
.
```

2.7. Gráficos

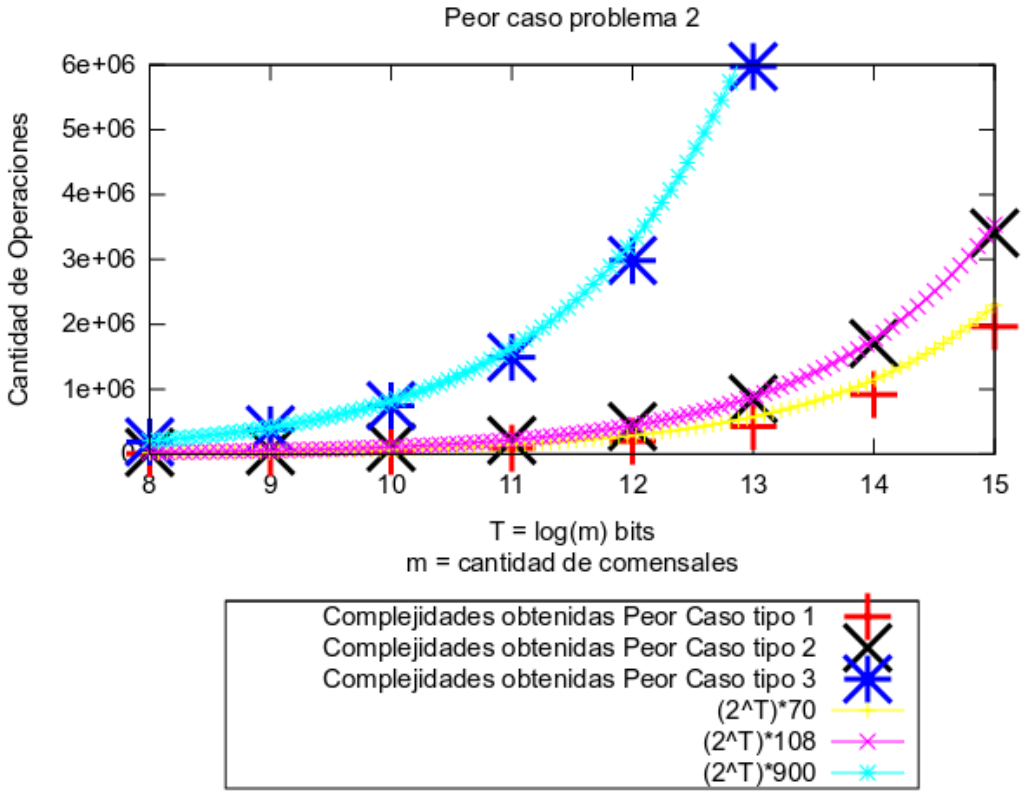


En el gráfico aparecen marcadas con X rojas los peores casos para tamaños de entradas desde 8 bits a 15 bits. La función teórica aumentada 70 veces logra acotar bastante bien la cantidad de operaciones del algoritmo. El problema es exponencial con respecto al tamaño de la entrada. Los gráficos siguientes muestran similares conclusiones variando la constante mutiplicativa.





En general, los tres gráficos muestran los peores casos en donde las mediciones obtenidas concuerdan con la complejidad teorica que habiamos calculado. Se puede observar también que las constantes utilizadas para comparar las mediciones con la función son mucho menores de lo que habiamos calculado en un primer momento como 2^{26}. Esto puede deberse a que si un comensal no es marcado en una decisión sobre un ingrediente, puede ser marcado en la contraria achicando el conjunto de comensales a evaluar.



En este último gráfico se puede comparar las mediciones de los tres tipos de casos juntas. Se puede ver que efectivamente el caso $Tipo3$ es el peor de todos ya que el primer comensal hace participar todos los ingredientes y nunca se filtran los comensales hasta llegar al último ingrediente.

2.8. Conclusiones

Luego de estudiar bastante el problema, y tratando de utilizar las nuevas técnicas aprendidas hasta el momento, llegamos a la conclusión que atacar el problema con *Backtracking* era lo más lógico.

Pensamos en un algoritmo *Goloso*, pero cualquier decisión arbitraria que tomáramos sobre algún ingrediente en particular podía ser facilmente rota poniéndo un participante con la preferencia contraria unicamente.

Con *Divide and Conquer* podríamos haber dividido la cantidad de comensales, pero podríamos haber llegado a subsoluciones incompatibles al momento del merge, mientras que una solución geneal era posible.

Con Programación Dinámica teníamos un problema parecido al de *Divide and Conquer*. Por último la idea de hacer un algoritmo que fuera construyendo una solución con la posibilidad de completarla antes de terminar de evaluar todos los ingredientes fue la que más se adecuó con la

idea que teniamos del problema pudiendo además mejorar la construcción por medio de podas fáciles de chequear en el momento indicado y que acortasen el recorrido por el árbol de decisión en una gran medida.

Fue bastante difícil encontrar la forma de la entrada para los peores casos, ya que teniamos que tener en cuenta que ésta tenía que evitar las podas y tratar de recorrer cada nodo del árbol de decisión con la mayor cantidad de comensales insatisfechos como fuera posible.

Esto nos dió la pauta que la técnica elegida fue la correcta.

3. Problema3: Conjetura de Goldbach

3.1. 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 y que 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 y mejores casos.

3.2. 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:

$$1 + (n\text{-}1),\ 2 + (n\text{-}2),\ \ldots\ (n/2) + (n/2),\ \ldots\ (n\text{-}2) + 2,\ (n\text{-}1) + 1$$

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:

$$1 + (n\text{-}1),\ 2 + (n\text{-}2),\ \ldots\ (n/2) + (n/2)$$

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.

$$2 + (n\text{-}2),\ \ldots\ (n/2) + (n/2)$$

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.

3.3. Pseudo-código

Algorithm 3 Goldbah

```
for cada n1,n2 candidatos podados do
  if esPrimo(n1) y esPrimo(n2) then
    retornar n1, n1
  end if
end for
```

3.4. 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.

3.5. 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)               O(raiz(n1)) + O(raiz(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 $O(\sqrt{n})$.

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($\sqrt{(n)}$).

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

Como $O(\sqrt{(2 \cdot i + 1)})$ y $O(\sqrt{n - (2 \cdot i + 1)}$ son acotables por $O(\sqrt{n})$.

Acotando:

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

$$\sum_{i=1}^{n/4} 2 \cdot \sqrt{n}$$

$$2 \cdot \sum_{i=1}^{n/4} \sqrt{n}$$

$$2 \cdot \frac{n^2}{4} \cdot \sqrt{n}$$

$\frac{n^{\frac{3}{2}}}{2}$ que es $O(n^{1,5})$

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 $O(n^{1,5})$ aplicamos cuadrado a ambos lados para ver la complejidad en T:

$$(2^T)^{1,5} = n^{1,5}$$

$$(2^{1,5T}) = n^{1,5}$$

$(2^{1,5T})$ es $O(2^T)$

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

3.6. Tests

Analizando el algoritmo es fácil describir porque un cierto valor da un peor o mejor caso, pero lo que no es tan sencillo es determinar cuáles con exactitud son aquellos valores o como agruparlos.

Peor Caso: Para determinar los sumandos primos se comienza por recorrer desde el menor número primo impar, el tres, tomando su complemento para con el n par, es decir $n-3$, y verificando si ambos son primos. De ser cierto ya habrá encontrado una solución. En caso contrario deberá seguir con el próximo... ¿Hasta cuándo?

El algoritmo tiene condición de corte cuando llega hasta la mitad de n. En el peor de los casos tendrá que llegar a esa mitad para encontrar, en caso que exista, ambos sumandos primos.

El peor caso entonces es cuando para un n par dado, el sumando más chico están lo más cerca de la mitad posible.

Mejor Caso: El mejor caso por el contrario es cuando para el número n uno de sus sumandos es 3, el primo impar mas pequeño.

En base a estos casos realizamos gráficos con que se pudiéra contrastar la teoría. Es por eso que decidimos graficar primeramente para cada número par cual sería el mínimo sumando primo. Las mediciones corresponden al gráfico *Menores Sumandos Primos*. Allí se aprecia que la mayoría de las soluciones contienen al 3, 5 y 7 cuyas ocurrencias aparecen con frecuencia.

Relizando una cantidad de mediciones mayor nos dimos cuenta que esto también parece continuar valiendo para una cantidad mayor de valores de entrada en el gráfico *Menores Sumandos Primos* donde los valores llegan a 1,000,000.

Parecería ser que a un mayor número, la base de sumandos primos pequeños se mantiene constante, pero a su vez aparecen más soluciones correspondientes a números más altos, aumentando así la cantidad de soluciones.

En el gráfico de *Cantidad de soluciones* podemos ver que a medida que los valores aumentan, efectivamente aumentan la cantidad de soluciones.
¿Será posible que la mayoría de estas soluciones sean çasos pequeñosz se conviertan estos en los casos promedio?

Dando un vistazo a las gráficas de *Complejidades Conjetura de Goldbach*, vemos que para cada entrada en bits tenemos varias mediciones de cantidad de operaciones (marcadas con signo

«+» y que aparecen verticalmente), que corresponden a cada valor de n dentro de ese rango de bits. Sucede la mayoria está contentrada hacia la mitad inferior y esto indica que le toma menos operaciones al algoritmo encontrar la solución. Esto solo puede deberse a que la solución contiene a un primo relativamente pequeño. Parecería ser que la pregunta anterior tendería a responderse con un si.

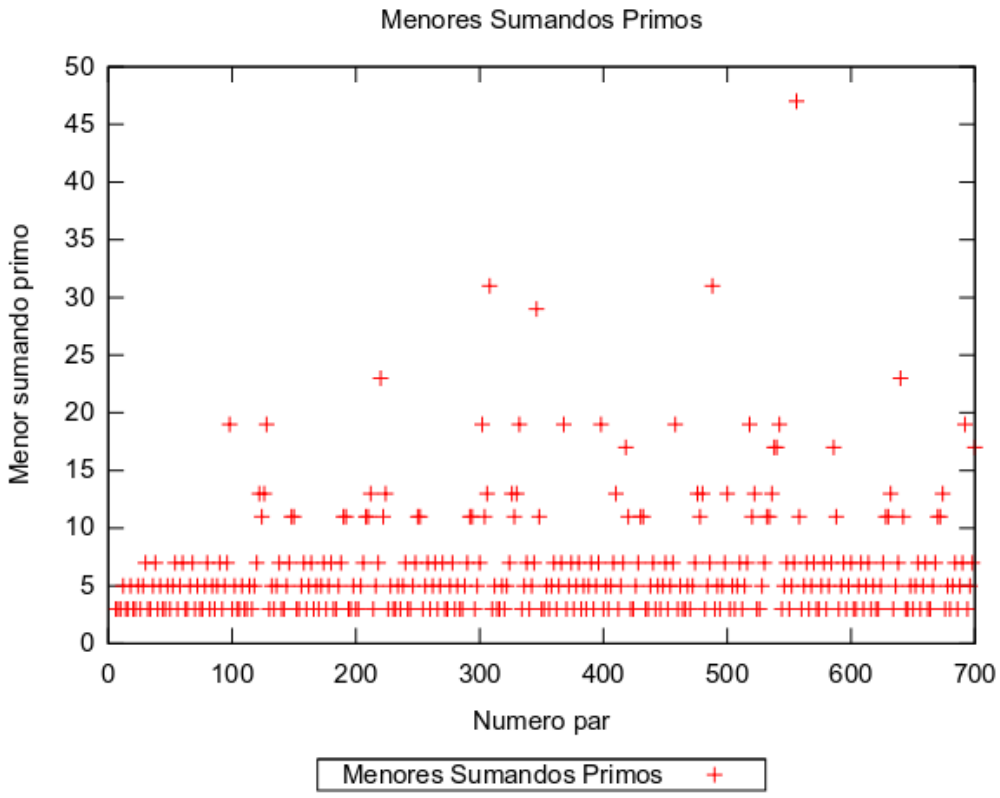
Pero de todos modos ¿Porqué hay tanta cantidad de operaciones, o signos -çoncentrados hacia abajo en el gráfico? ¿No deberían estos casos pequeños bajar aún más la cantidad de operaciones?

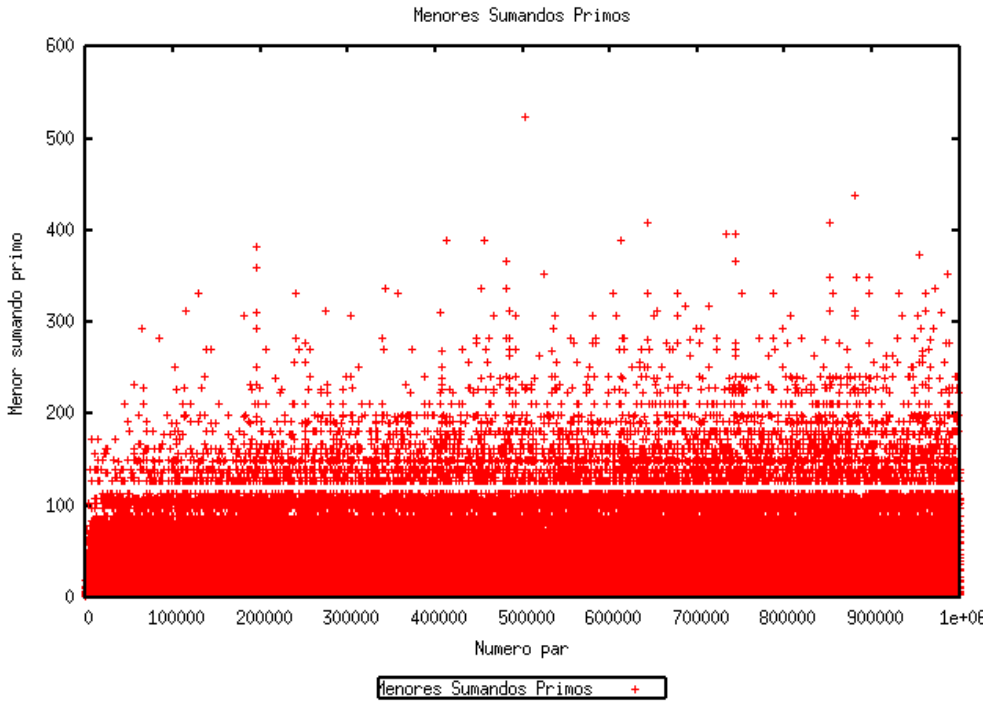
Una explicación es que por más que las solución contenga un primo relativamenteçhico, hasta llegar a saber que él es solución, se verificaron primero todos los sumandos anteriores para validad si ellos eran solución. Todas estas verificaciones consumieron operaciones que terminaron por afectar la solución final. Es por ello que finalmente la complejidad es exponenciál al tamaño de la entrada.

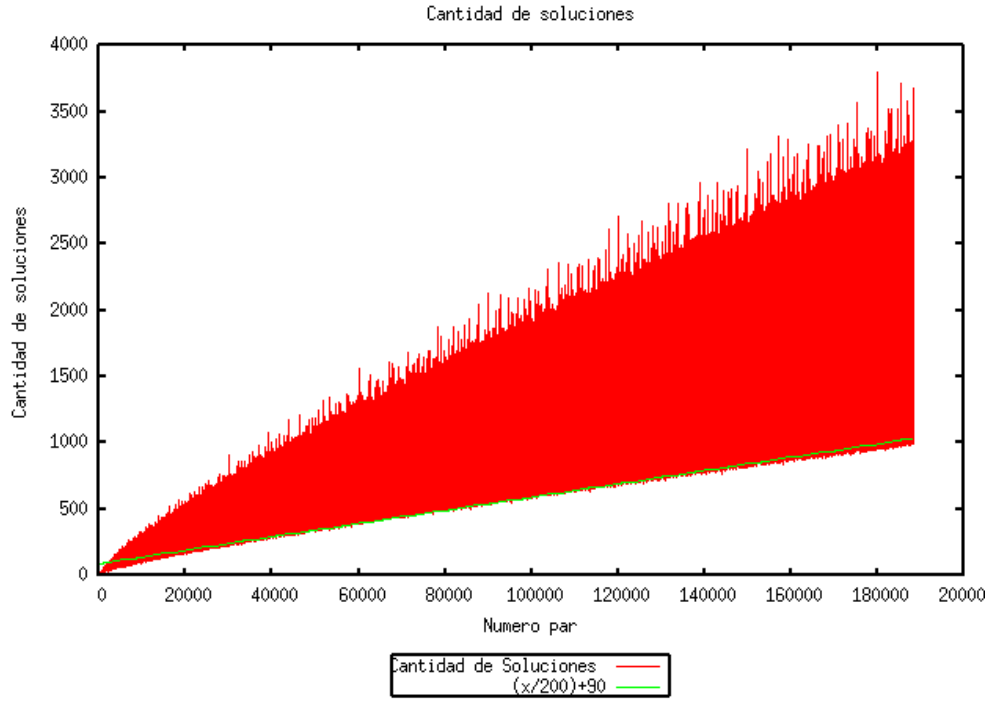
Finalmente terminamos entendiendo que cada número primo rige como solución para una serie de números pares de entrada. Podríamos de esta forma, agrupar todos los pares en *familias* dependiendo de los sumandos primo que formen parte en su solución. Estas familias podían solaparse ya que el 3 y el 5 podrían formar parte, juntos o separados de una solución. por ejemplo la del 8.

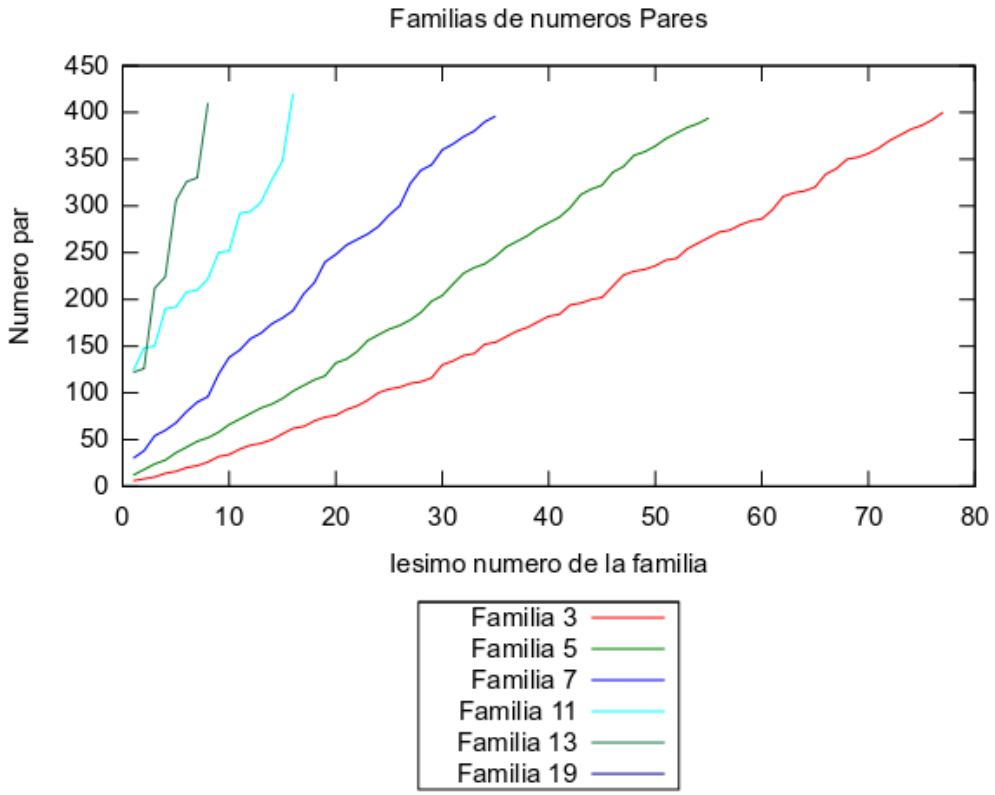
A tal efecto realizamos mediciones para obtener el gráfico *Familia de numeros Pares* en donde muestra cual es el i-esimo numero par que es solución de cada familia. La idea de esta medición es ver la incidencia que tienen los primos en las soluciones viendo por ejemplo como números primos mayores incidirán mas tarde, es decir, en las soluciones de números mayores.

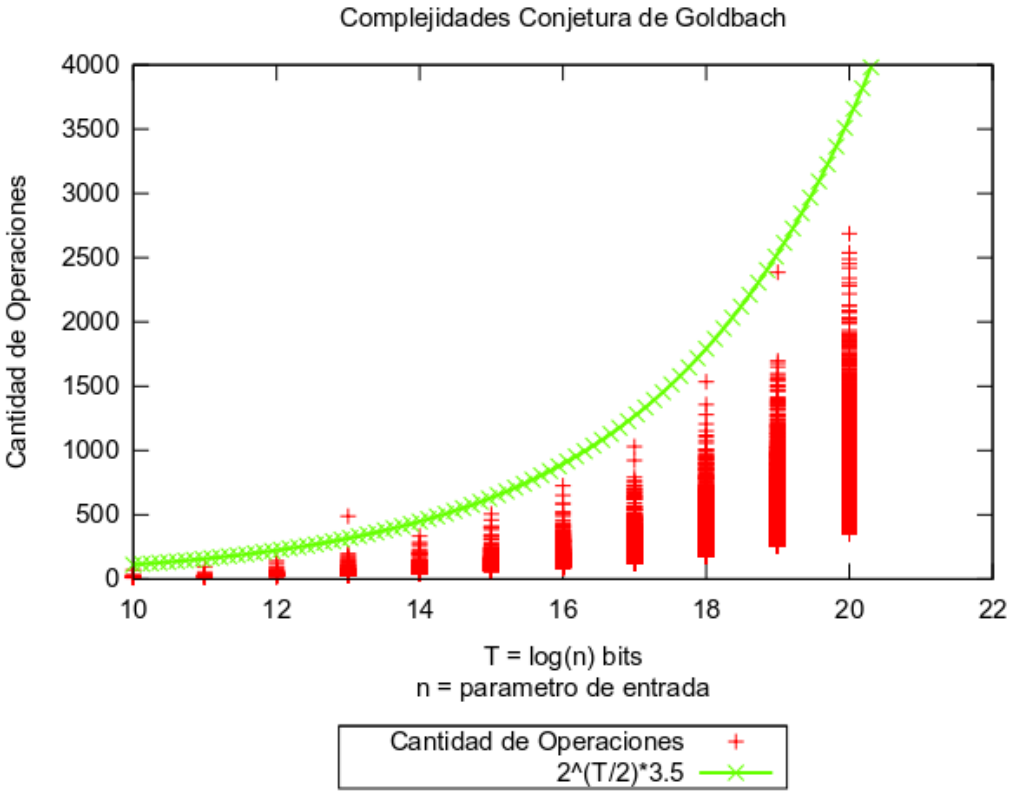
3.7. Gráficos

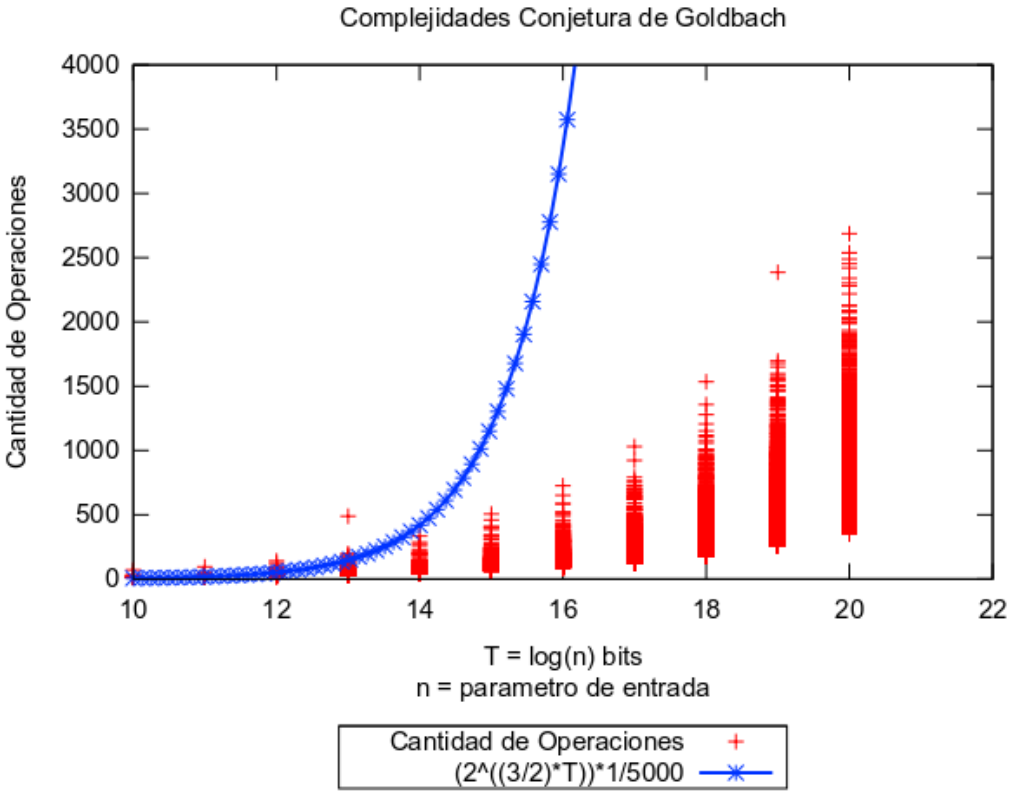

Menores Sumandos Primos
Menor sumando primo
Numero par
0
5
10
15
20
25
30
35
40
45
50
0
100
200
300
400
500
600
700
Menores Sumandos Primos

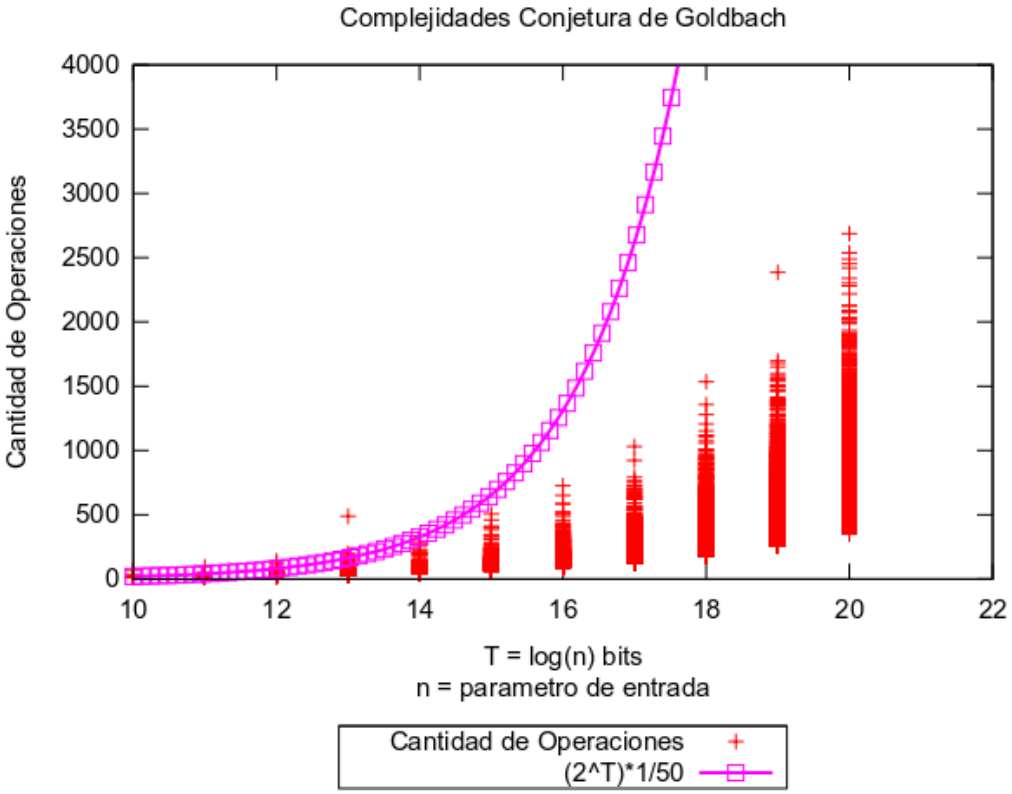

Menores Sumandos Primos
Menor sumando primo
Numero par
0
100
200
300
400
500
600
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
Menores Sumandos Primos

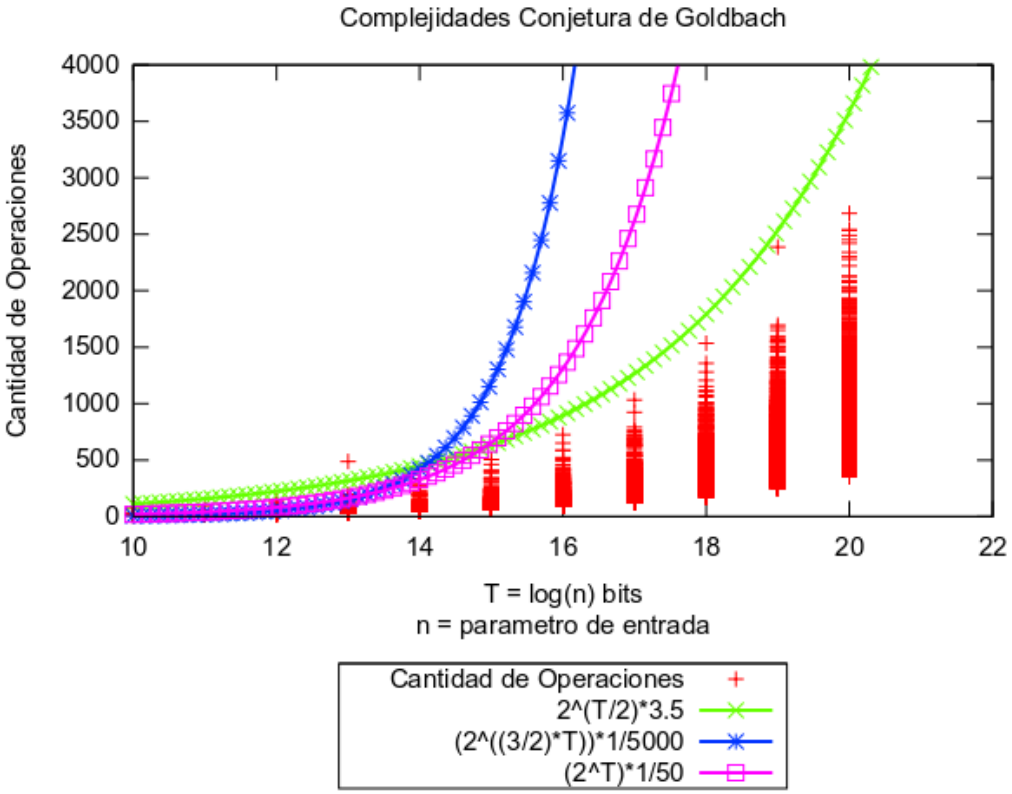

Cantidad de soluciones
Cantidad de soluciones
Numero par
0
500
1000
1500
2000
2500
3000
3500
4000
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
20000
Cantidad de Soluciones
(x/200)+90


Familias de numeros Pares
Numero par
Iesimo numero de la familia
0
50
100
150
200
250
300
350
400
450
0
10
20
30
40
50
60
70
80
Familia 3
Familia 5
Familia 7
Familia 11
Familia 13
Familia 19


Complejidades Conjetura de Goldbach
Cantidad de Operaciones
4000
3500
3000
2500
2000
1500
1000
500
0
10
12
14
16
18
20
22
T = log(n) bits
n = parametro de entrada
Cantidad de Operaciones
2^(T/2)*3.5


Complejidades Conjetura de Goldbach
Cantidad de Operaciones
4000
3500
3000
2500
2000
1500
1000
500
0
10
12
14
16
18
20
22
T = log(n) bits
n = parametro de entrada
Cantidad de Operaciones
(2^((3/2)*T))*1/5000


Complejidades Conjetura de Goldbach
Cantidad de Operaciones
4000
3500
3000
2500
2000
1500
1000
500
0
10
12
14
16
18
20
22
T = log(n) bits
n = parametro de entrada
Cantidad de Operaciones
(2^T)*1/50


Complejidades Conjetura de Goldbach
Cantidad de Operaciones
4000
3500
3000
2500
2000
1500
1000
500
0
10
12
14
16
18
20
22
T = log(n) bits
n = parametro de entrada
Cantidad de Operaciones
2^(T/2)*3.5
(2^((3/2)*T))*1/5000
(2^T)*1/50

3.8. Conclusiones

Fue interesante notar que si bien para un entero en particular el algorimo realiza X cantidad operaciones, medir en función de los bits entrada hace que para una determinada cantidad de ellos se correspondan varios valores enteros y que haga que la cantidad de operaciones varíe aún cuando el tamaño de entrada sea el mismo.

Los problemas relacionados con primos siempre son interesantes porque nada pareciera regirlos con exactitud, y de esta manera se pueden elaborar teorías y conjeturas con las cuales entretenerse.