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.

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

1.2. Como ejecutar el TP

Mediante Eclipse

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

Desde el Eclipse

- 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:

- problema1

  Torneo.java

- problema2

  Pizza.java

- problema3

  Goldbach.java

2. Problema1: Torneo

2.1. Introducción

Debemos organizar un torneo con cierta cantidad de competidores que la llamaremos n. 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.

Para ello, implentaremos un algoritmo que utilice la técnica de divide y vencerás. En cada llamada recursiva, el problema se divide la cantidad de jugadores por 2, se calcula los partidos que juegan entre sí de la primera mitad de ellos, y luego se combinan esos resultados para la otra mitad de los competidores.

2.2. 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, para ir guardando la tabla de como van quedando los partidos entre cada competidor, 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 de las subsoluciones, que termina calculando las competiciones para esta 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 generó 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 que otorgaró a ese jugador el oponente en ese día, o caso contrario, implicaría que el jugador ese 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.

2.3. Pseudo-código

```
if cantJugadores == 2 then
  fixturePartidos[1][1] = 2;
  //caso base
  fixturePartidos [2][1] = 1;
  //caso base
else
  if (cantJugadores mod 2)!=0 then
    //cantidad jugadores impar
    torneo(cantJugadores+1); //recursion con jugador ficticio
    colocarJugadorFicticio(cantJugadores); //pone en cero a ese jugador
  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
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
  cuadranteInferiorDerecho(mitadJugadores, cantJugadores, esPar)
  if esPar then
    for cada jugador entre mitadJugadores+1; y cantJugadores do
      for cada dia entre mitadJugadores y cantJugadores-1 do
        if jugador¿dia then
          fixturePartidos[jugador][dia]=jugador-dia;
        else
          fixturePartidos[jugador][dia] = (jugador + mitadJugadores)-dia;
        end if
      end for
    end for
  else
    for cada jugador entre 1 y mitadJugadores do
      for cada dia entre mitadJugadores+1; y cantJugadores-1 do
        if jugador+dia ¡=cantJugadores then
          fixturePartidos[jugador][dia] = jugador+dia;
        else
          fixturePartidos[jugador][dia] = jugador + dia - mitadJugadores;
        end if
      end for
    end for
  end if
  cuadranteSuperiorDerecho(mitadJugadores, cantJugadores, esPar)
  if esPar then
    for cada jugador entre 1 y mitadJugadores do
      for cada dia = mitadJugadores; dia ¡= cantJugadores-1; dia++ do
        if (jugador+dia)¡=cantJugadores) then
          fixturePartidos[jugador][dia]=jugador+dia;
        else
          fixturePartidos[jugador][dia]=jugador + dia - mitadJugadores; //NO METER UN
          FICTICIO
        end if
      end for
    end for
  else
    for cada jugador entre mitadJugadores+1 y cantJugadores do
      for cada dia = mitadJugadores+1; dia ¡= cantJugadores-1; dia++ do
```

```
        if jugador¿dia then
          fixturePartidos[jugador][dia] = jugador -dia;
        else
          fixturePartidos[jugador][dia] = (jugador + mitadJugadores)-dia;
        end if
      end for
    end for
  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
```

2.4. Complejidad

```
  if cantJugadores == 2 then
    fixturePartidos[1][1] = 2; //caso base
    fixturePartidos [2][1] = 1; //caso base
  else
    if (cantJugadores mod 2)!=0 then
      //cantidad jugadores impar
      torneo(cantJugadores+1); //recursion con jugador ficticio
      colocarJugadorFicticio(cantJugadores); //pone en cero a ese jugador
    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
  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
  cuadranteInferiorDerecho(mitadJugadores, cantJugadores, esPar)
  if esPar then
    for cada jugador entre mitadJugadores+1; y cantJugadores do
      for cada dia entre mitadJugadores y cantJugadores-1 do
        if jugador¿dia then
          fixturePartidos[jugador][dia]=jugador-dia;
        else
          fixturePartidos[jugador][dia] = (jugador + mitadJugadores)-dia;
        end if
      end for
    end for
  else
    for cada jugador entre 1 y mitadJugadores do
```

```
      for cada dia entre mitadJugadores+1; y cantJugadores-1 do
        if jugador+dia ¡=cantJugadores then
          fixturePartidos[jugador][dia] = jugador+dia;
        else
          fixturePartidos[jugador][dia] = jugador + dia - mitadJugadores;
        end if
      end for
    end for
  end if
  cuadranteSuperiorDerecho(mitadJugadores, cantJugadores, esPar)
  if esPar then
    for cada jugador entre 1 y mitadJugadores do
      for cada dia = mitadJugadores; dia ¡= cantJugadores-1; dia++ do
        if (jugador+dia)¡=cantJugadores then
          fixturePartidos[jugador][dia]=jugador+dia;
        else
          fixturePartidos[jugador][dia]=jugador + dia - mitadJugadores; //NO METER UN
          FICTICIO
        end if
      end for
    end for
  else
    for cada jugador entre mitadJugadores+1 y cantJugadores do
      for cada dia = mitadJugadores+1; dia ¡= cantJugadores-1; dia++ do
        if jugador¿dia then
          fixturePartidos[jugador][dia] = jugador -dia;
        else
          fixturePartidos[jugador][dia] = (jugador + mitadJugadores)-dia;
        end if
      end for
    end for
  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 en dividir.

Si analizamos la combinación, veamos primero cual es el orden de cuadranteInferiorDerecho. 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 es de O ((n / 2) al cuadrado)

El análisis para cuadranteInferiorIzquierdo, y cuadranteSuperiorDerecho es anáalogo el de cuadranteInferiorDerecho, pero al cuadranteInferiorIzquierdo hay que sumarle el tiempo que se demora en colocar el jugador ficticio, que es de O ((n / 2) al cuadrado), ya que nuevamente, recorre n / 2 jugadores y n/2 días.

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

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

Por último, se debe dar el resultado (que se guarda en un archivo) de como quedo creado el torneo, por lo que se debe recorrer la matriz ya completa, e ir copiando en el archivo los resultados. Esa iteración queda nuevamente O (n al cuadrado) por lo que, el algoritmo se demora en total, O((n al cuadro) * log(n)) + n al cuadrado))

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 a la T.

Entonces, la complejidad del algoritmo en función del tamaño de entrada es de O((log 2 a la T) * 2 a la T cuadrado + 2 a la T cuadrado).

2.5. Tests

Los test se han realizado hasta 5000 competidores de diferentes formas. En ellos hemos notado que obtuvimos un mejor caso en nuestro algoritmo, que es cuando la cantidad de jugadores es potencia de 2, y otro que pareciera ser el peor caso, cuando la cantidad de competidores es de la forma ((2 a la i) + 1), ya que tiene que agregar y sacar jugadores «ficticios» en una o más veces.

2.6. Gráficos

2.7. Conclusiones

3. Problema2: Pizza entre amigos

3.1. 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 no se puede 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, o responder que no es posible.

3.2. 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 comenzales un ingrediente repetido. Se descartaran las apariciones repetidas de ese ingrediete en caso que la decisión tomada sobre el mismo sea la misma EJ +A+B+A lo tomaremos como +A+B. si alguna de las repeticiones tiene la desición contrária (EJ +A+B-A), entonces asumiremos que este comenzal está satisfecho por defecto 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 comenzal ya que este no tiene preferencias.

Fuera de estos casos, asumimos que todos los comenzales 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 arbol 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 comenzales que fueron satisfechos con la desición tomada sobre el ingrediente actual, cuando se llega al último ingrediente si todos los comenzales 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 comenzales 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 comenzales que no hayan sido marcados ya en la solución parcial actual, ya que los demas están satisfechos y seguiran estandolo cualquiera sea la desición que se tome sobre el resto de la pizza.

Poda3: Antes de armar el arbol se revisa las preferencias de cada uno de los comenzales 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 está de mas revisarlo.

Poda4: Como estamos recorriendo el arbol de decisión desde la primera hasta la última letra (ingrediente) en orden, podemos saber cuando un comenzal 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 comenzal.

Tip: Vamos marcando los comenzales que fueron satisfechos junto con que ingrediente fueron satisfechos, por si hay que volver en el arbol se desmarcan solo esos y se prueba otra solución

3.3. Pseudo-código

Algorithm 1 Pizza

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

```
backtrack(i, ponerEnLaPizza)
if ponerEnLaPizza == true then
  poner i en la pizza
end if
marcar comenzales satisfechos
```

```
if no hay mas comenzales 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 comenzales satisfechos
  if ponerEnLaPizza == true then
    sacar i de la pizza
  end if
  return false
end if
```

3.4. Modelo Elejido

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

3.5. Complejidad

n = Cantidad de Ingredientes
m = Cantidad de Comezales Preguntar si un comenzal prefiere un ingrediente es $O(1)$
Preguntar si quedan comenzales insatisfechos es $O(1)$
Armar la solución es $O(n)$
Mover un comenzal de una lista a otra es $O(1)$
(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 comenzales 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 comenzal puede llegar a tener hasta 26 preferencias, estos son 52 caracteres por comenzal
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:
$m = 2^T$

3.6. Tests

Tipo1: Un estudio poco profundo nos diría que los peores casos son aquellos que tiene todas las permutaciones. A estos los llamaremos $Tipo1$
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: Evaluando un poco mejor el algoritmo pudimos observar que los peores casos para cada tamaño de entrada los tendrá cuando tenga que evaluar en la mayoria de los comenzales en cada ingrediente.
Sabiendo que el algoritmo primero intenta poner el ingrediente y luego, en caso que no haya solucion prueba no poniendolo lo obligamos a hacer el mayor numero de operaciones por la rama true.
Tambien tenemos que procurar que no termine hasta haber revisado el último ingrediente asi que ponemos algun comenzal que tenga una preferencia contraria. Si ese comenzal tiene mas de una preferencia entonces se podria satisfacer evitando que se revisen mas ingredientes asi que no le ponemos mas. Los peores casos serian 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 evalue la mayor cantidad posible de comenzales en cada nodo. La solución fue esta:

```
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;
.
```

3.7. Gráficos

Se puede ver en los tres gráficos de peores casos que las mediciones obtenidas concuerdan con la complejidad teorica que habiamos calculado. Tambien se puede observar 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 comenzal no es marcado en una decisión sobre un ingrediente, puede ser marcado en la contraria achicando el conjunto de comenzales a evaluar. En este ultimo gráfico se puede comparar las mediciones de los tres tipos de casos juntas. Se puede ver que efectivamente el caso *Tipo*3 es el peor de todos ya que el primer comenzal hace participar todos los ingredientes y nunca se filtran los comenzales hasta llegar al último ingrediente.

3.8. 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

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

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