Práctica 1: Divide y vencerás

Rubén Dugo Martín
Profesor: Rafael Alcalá
Teoría de Algoritmos
Curso 2009-2010
Universidad de Granada

Índice de contenido

1. Rango de dominancia

1.1 Algoritmo Iterativo

Descripción en pseudocódigo

```
Procedimiento Iterativo (v,r: vector)

      Para i=0 hasta tamaño(v)-1
            Para j=0 hasta tamaño(v)-1
                  Si (( i!=j) y (v[i].x>v[j].y) y
(v[i].y>v[j].y))
                        r[i] ← r[i]+1
                  Fin Si
            Fin Para
      Fin Para

Fin Procedimiento
```

Análisis teórico

```
(1)     void iter_rango (const vector < pair<float,float> > S,
  vector<float> &rango)
(2)     {
(3)         const unsigned int size = S.size();
(4)
(5)         rango.resize(size);
(6)
(7)         for (unsigned int i=0;i<size;i++)
(8)         {
(9)             rango[i] = 0.0;
(10)            for (unsigned int j=0;j<size;j++)
(11)            {
(12)                if (i!=j &&
(13)                    S[i].first > S[j].first &&
(14)                    S[i].second > S[j].second)
(15)                        rango[i]++;
(16)            }
(17)        }
(18)    }
```

Obviando instrucciones insignificantes paso a analizar bucle contenido en las líneas 7-17, claramente el bucle interno se repetirá **n** veces (desde 0 hasta size=n) y el bucle exterior **n** veces también (idem). Las instrucciones del bucle interior las tomamos como constantes *O(1)* al igual que la del bucle exterior.

Por lo que tenemos $T(n) = n \cdot n = n^2 \in O(n^2)$.

Análisis empírico

Con la finalidad de realizar el análisis empírico de todas las implementaciones de este apartado he realizado el siguiente módulo:

```
#include <time.h>
#include <stdlib.h>
#include <iostream>
#include <map>
#include <vector>



using namespace std;


void iter_rango (const vector < pair<float,float> > S, vector<float>
&rango);
void dyv_rango_1 (const vector < pair<float,float> > S, vector<float>
&rango, const int umbral);
void dyv_rango_2 (const vector < pair<float,float> > S, vector<float>
&rango, const int umbral);


void rellena(vector<pair<float, float> > &v, const unsigned int &n)
{
    v.resize(n);
    for (unsigned int i=0;i<n;i++)
    {
        v[i].first = (float) rand()/RAND_MAX;
        v[i].second = (float) rand()/RAND_MAX;
    }
}


(...)
```

```
int main(int argc, char **argv)
{
    vector<float> rango;
    vector<pair<float, float> > v;
    const unsigned int FASES=15;
    double tiempos[3][FASES];

    clock_t antes,despues;
    unsigned int i,j,n=1,veces;
    const unsigned int NUM_PEQ=100000, NUM_GRANDE=1;

    srand(time(NULL));

    for (j=0;j<12;j++)
      {
        veces = ((j>5)?NUM_GRANDE:NUM_PEQ);

        rellena(v, n);

        fprintf(stderr, "%d iter\n",n);
        antes = clock ( ) ;
        for ( i =0 ; i<veces; i++)
        {
            iter_rango (v, rango);
        }
        despues = clock ( ) ;
        tiempos[0][j] = ( ( double ) ( despues-antes) / (double)
(CLOCKS_PER_SEC * veces) );


            fprintf(stderr, "%d dyv1\n",n);
            antes = clock ( ) ;
            for ( i =0 ; i<veces; i++)
            {
                  dyv_rango_1 (v, rango, 1);
        }
        despues = clock ( ) ;
            tiempos[1][j] = ( ( double ) ( despues-antes) / (double)
(CLOCKS_PER_SEC * veces) );
(...)
```

```
        fprintf(stderr, "%d dyv2\n",n);
            antes = clock ( ) ;
            for ( i =0 ; i<veces; i++)
            {
                  dyv_rango_2 (v, rango, 1);
        }
        despues = clock ( ) ;
            tiempos[2][j] = ( ( double ) ( despues-antes) / (double)
(CLOCKS_PER_SEC * veces) );

            n = n * ((j%2)?2:5);
      }

      for(i=0;i<3;i++)
      {
          n=1;
          for(j=0;j<FASES;j++)
              cout<<n<<' '<<tiempos[i][j]<<endl;

        n = n * ((j%2)?2:5);
    }
}
```

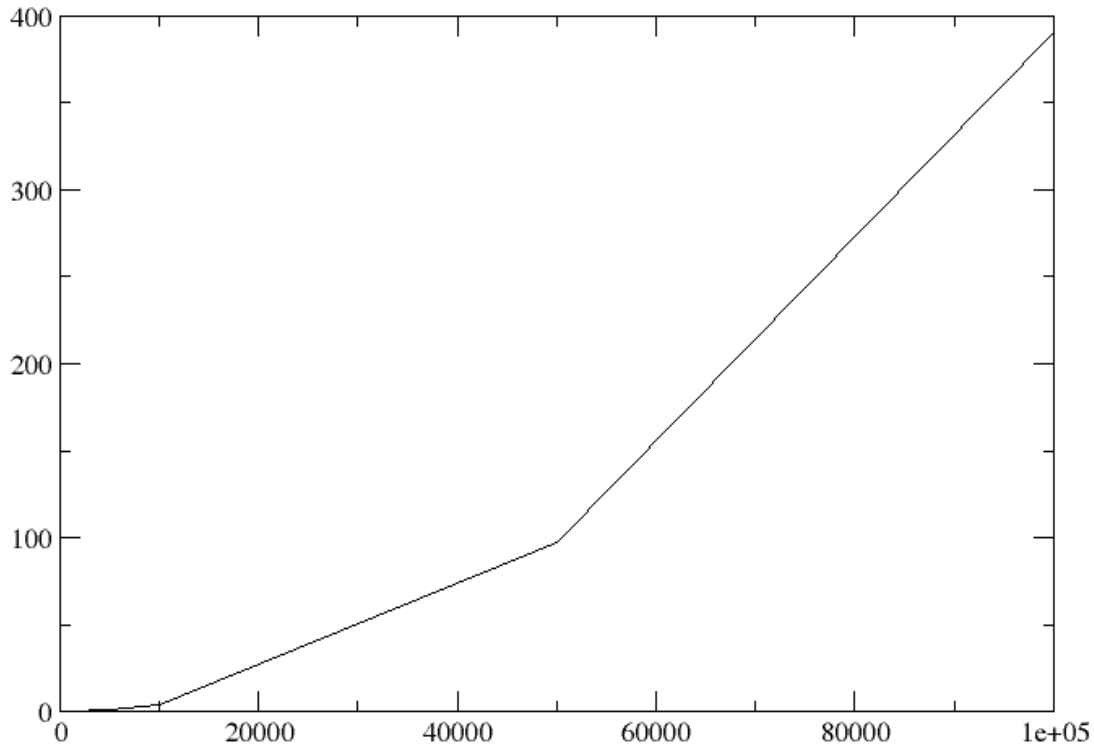
El cual me hace un incremento de *n* desde 1 hasta 100000, multiplicando n por 2 si j es impar o por 5 si j es par, esto quiere decir que *n* va de la fomar: 1, 5, 10, 50… hasta 100000.

También imprime por la salida de error por que valor va y en que algoritmo se encuentra, para de esta forma tener un traza normal de lo que ocurre.

En los ficheros adjuntos se pueden ver los resultados.

Análisis híbrido

Llevando los datos obtenidos en el apartado anterior obtenemos la siguiente gráfica:



Introduciendo como fórmula la siguiente: y = *a0*x^2 + a1* obtengo los siguientes parámetros:

a1 = 3.90586·10^{-8}

a2 = -2.41609·10^{-16}

Y coeficiente de relación *1*, perfecto.

1.2 Algoritmo DyV

1.2.1 Versión 1: ordenando por y en cada paso

Descripción en pseudocódigo

```
Procedimiento DyV_1 (v,r: vector; umbral: entero)

      x, y, posx, posy, rango_desordenado: vector

      Para i=0 hasta tamaño(v)-1
            x[i] ← v[i].x
            posx[i] ← i
            rango_desordenado[i] ← 0.0
      Fin Para

      Quicksort (x, posx)

      Para i=0 hasta tamaño(v)-1
            y[i] ← v[posx[i]].y
            posy[i] ← i
      Fin Para

      DyV_1_rec (y, posy, rango_desordenado, 0, tamaño(v)-1,
umbral)

      Para i=0 hasta tamaño(v)-1
            rango[posx[i]] ← rango_desordenado[i]
      Fin Para

Fin Procedimiento
```

```
Procedimiento DyV_1_rec (y, posy, r: vector; ini,fin,umbral:
      entero)

      posmedia: entero
      puntosS1: flotante

      Si (fin-ini+1 <= 1) Entonces
            Retorno
      Fin Si

      Si (fin-ini+1 <= umbral) Entonces
            Para i=ini hasta fin
                  Para j=ini hasta i-1
                        Si (y[i] > y[j]) Entonces
                              r[i] ← r[i]+1
                        Fin Si
                  Fin Para
            Fin Para

            Retorno
      Fin Si

      posmedia ← (ini+fin)/2
      puntosS1 ← 0.0

      DyV_1_rec (y, posy, r, ini, posmedia, umbral)
      DyV_1_rec (y, posy, r, posmedia+1, fin, umbral)

      Quicksort (y, posy)

      Para i=ini hasta fin
            Si (posy[i] > posmedia) Entonces
                  rango[posy[i]] ← rango[posy[i]] + puntosS1
            Si No
                  puntosS1 ← puntosS1+1
            Fin Si
      Fin Para

Fin Procedimiento
```

Análisis teórico

Comenzamos analizando la función *dyv_rango_1* que actúa de interfaz para la función recursiva:

Nota: Todos los logaritmos expresados en este apartado son en base 2.

```
(19)  void dyv_rango_1 (const vector < pair<float,float> > S,
      vector<float> &rango, const int umbral)
(20)  {
(21)      unsigned int i;
(22)      const unsigned int size=S.size();
(23)
(24)      vector<float> x(size),y(size), rango_desordenado(size);
(25)      vector<unsigned int> posx(size), posy(size);
(26)
(27)      for (i=0;i<size;i++)
(28)      {
(29)          x[i] = S[i].first;
(30)          posx[i] = i;
(31)          rango_desordenado[i] = 0.0;
(32)      }
(33)
(34)      dyv_rango_qs(x,posx,0,size-1);
(35)
(36)      for (i=0;i<size;i++)
(37)      {
(38)          y[i] = S[posx[i]].second;
(39)          posy[i] = i;
(40)      }
(41)
(42)      dyv_rango_1_rec(y, posy, rango_desordenado, 0, size-1, umbral);
(43)
(44)      rango.resize(size);
(45)
(46)      for (i=0;i<size;i++)
(47)      {
(48)          rango[posx[i]] = rango_desordenado[i];
(49)      }
(50)  }
```

Lo contenido en las línea 3-7 lo tomamos como operaciones constantes, el bucle situado en las línea 9-14 cuyo contenido es constate se realizará **n** veces, la llamada a la función *dyv_rango_qs* (línea 16) consumirá un tiempo **n·log(n)**, ya que su implementación asegura esto.

El bucle localizado en las líneas 18-22 cuyo contenido es constate se ejecutará **n** veces, el tiempo invertido en la llamada a la función recursiva (línea 24) lo llamaremos $\mathbf{f_1(n)}$, obviamos la línea 26 y finalmente el bucle (con contenido constante) situado en las líneas 28-31 se ejecutará **n** veces.

Sumando todos los tiempos obtenemos $\boldsymbol{T(n) = 3 \cdot n + n \cdot \log n + f_1(n)}$.

Ahora obtenemos el tiempo correspondiente a la función recursiva:

```
(1)   void dyv_rango_1_rec (vector<float> &y, vector<unsigned int> &posy,
vector<float> &rango, const unsigned int &ini, const unsigned int &fin,
const unsigned int &umbral)
(2)   {
(3)
(4)       if (fin-ini+1<=1) return;
(5)
(6)       if (fin-ini+1<=umbral)
(7)       {
(8)           dyv_rango_it(y, rango, ini, fin);
(9)           return;
(10)      }
(11)
(12)      unsigned int posmedia = (ini+fin)/2,i;
(13)      float puntosS1 = 0.0;
(14)
(15)      dyv_rango_1_rec(y, posy, rango, ini, posmedia, umbral);
(16)      dyv_rango_1_rec(y, posy, rango, posmedia+1, fin, umbral);
(17)
(18)      dyv_rango_qs(y, posy, ini, fin);
(19)
(20)      for (i=ini;i<=fin;i++)
(21)      {
(22)          if (posy[i] > posmedia)
(23)          {
(24)              rango[posy[i]]+=puntosS1;
(25)          }
(26)          else puntosS1++;
(27)      }
(28)  }
```

Obtenemos por la línea 4 que $\mathbf{f_1(0) = f_1(1) = 1}$.

Por la línea 6 deducimos que:

$$\mathbf{f_1(n) = n^2,\ n <= umbral}$$

Puesto que el tiempo de la función iterativa es $O(n^2)$.

El tiempo invertido en las instrucciones de la líneas 12 y 13 lo podemos despreciar.

En las líneas 15 y 16 se realizan dos llamadas recursivas cada una de ellas con *n/2* elementos, por lo que tenemos que $\mathbf{f_1(n) = 2 \cdot f_1(n/2) + f_2(n)}$.

Veamos el tiempo total invertido en las instrucciones que componen la función recursiva, $f_2(n)$;

- La llamada a la función *dyv_rango_qs* (línea 18) consumirá un tiempo **nlog(n)**.
- El bucle situado en las línea 22-29 cuyo contenido podemos considerar de orden constate se repetirá en el peor de los casos **n** veces, con *ini=0* y *fin=n-1*.

Finalmente obtenemos $\boldsymbol{f_1(n) = 2 \cdot f_1(n/2) + n \cdot log(n) + n}$ con $\boldsymbol{f_1(1) = 0}$, para **n > umbral**.

Resolvemos la recurrencia haciendo el cambio de variable $n = 2^m$;

$f_1(2^m) = 2 \cdot f_1(2^{m-1}) + 2^m \cdot m + 2^m$ *para* $m > log(umbral)$

$f_1(2^m) - 2 \cdot f_1(2^{m-1}) = 2^m \cdot m + 2^m$, *haciendo el cambio* $f_1(2^m) = t(m)$

$t(m) - 2 \cdot t(m-1) = 2^m \cdot m + 2^m$, *obtenemos la ecuación de recurrencia*

$x - 2 = 0$
$p(x) = (x - 2)(x - 1)^2$

cuyas raizes son: 2 y 1 (con multiplicidad 2)

la recurrencia queda como $t(m) = c_1\, 2^m + c_2\, 1^m + c_3\, m\, 1^m$
calculamos las 3 constantes sabiendo que: $t(0) = t(1) = log(1)$ *y* $t(2) = log(4)$

$t(0) = c_1 + c_2 = 0$
$t(1) = 2 \cdot c_1 + c_2 + c_3 = 0$
$t(2) = 4 \cdot c_1 + c_2 + 2 \cdot c_3 = 2$

resolvemos y obtenemos; $c_1 = 2, \quad c_2 = -2$ *y* $c_3 = 2$.
sustituimos: $t(m) = 2 \cdot 2^m + 2 \cdot m - 2$
ahora deshacemos el cambio de variable: $f_1(n) = 2 \cdot n + 2 \cdot n \cdot log n - 2$

Sustituyendo en la ecuación principal:

$\boldsymbol{T(n) = nlogn + n^2 + 3n,\ n <= umbral}$
$\boldsymbol{T(n) = 3nlogn + 5n - 2,\ \ n > umbral}$.

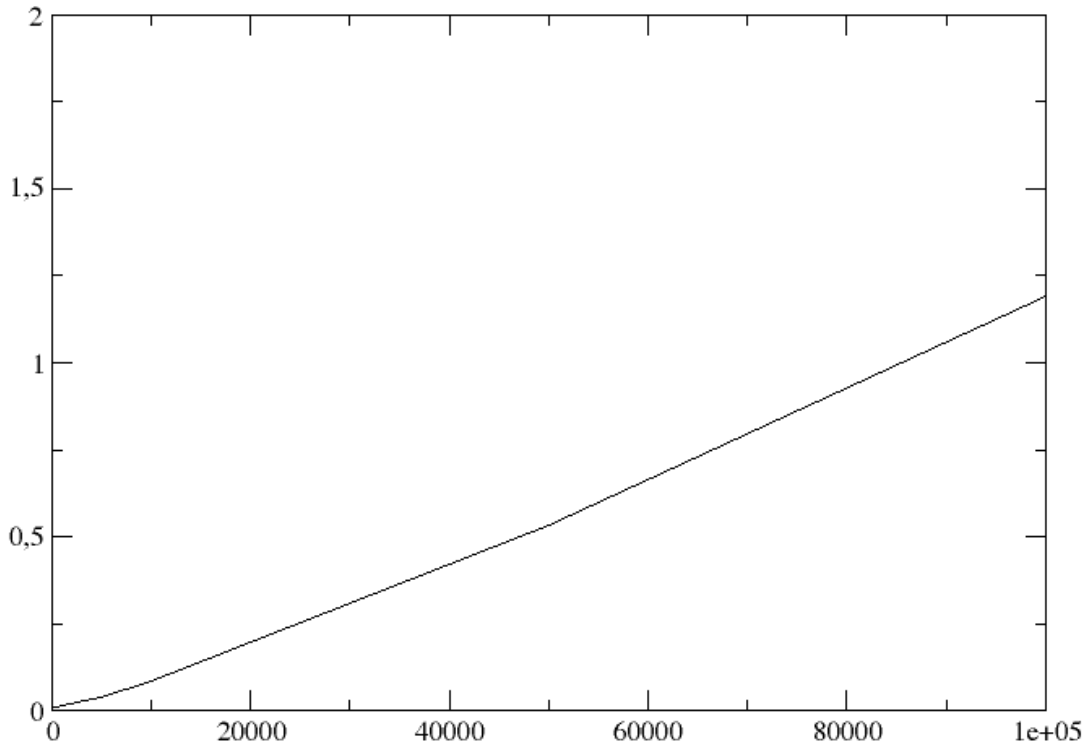
La función tendrá el orden $\boldsymbol{O(nlogn)}$ ***para n>umbral y*** $\boldsymbol{O(n^2)}$ ***en caso contrario***.

Análisis empírico

Visto en el apartado *1.1*.

Análisis híbrido

Llevando los datos obtenidos en el apartado anterior obtenemos la siguiente gráfica:



Introduciendo como fórmula la siguiente: y = *a0*x*log2(x) + a1*x + a2* obtengo los siguientes parámetros:

a0 = 1.37191·10^{-6}
a1 = -1.09984·10^{-5}
a2 = 0.010453

Y coeficiente de relación ***0.999721***, muy próximo a *1*.

1.2.2 Versión 2: ordenando por y sólo al principio

Descripción en pseudocódigo

```
Procedimiento DyV_2 (v,r: vector; umbral: entero)

      x, y, posx, posy, rango_desordenado: vector

      Para i=0 hasta tamaño(v)-1
            x[i] ← v[i].x
            posx[i] ← i
            rango_desordenado[i] ← 0.0
      Fin Para

      Quicksort (x, posx)

      Para i=0 hasta tamaño(v)-1
            y[i] ← v[posx[i]].y
            posy[i] ← i
      Fin Para

      Quicksort (y, posy)

      DyV_2_rec (y, posy, rango_desordenado, 0, tamaño(v)-1,
umbral)

      Para i=0 hasta tamaño(v)-1
            rango[posx[i]] ← rango_desordenado[i]
      Fin Para

Fin Procedimiento
```

```
Procedimiento DyV_2_rec (y, posy, r: vector; ini,fin,umbral:
      entero)

      posmedia: entero
      puntosS1: flotante

      Si (fin-ini+1 <= 1) Entonces
            Retorno
      Fin Si

      Si (fin-ini+1 <= umbral) Entonces
            Para i=ini hasta fin
                  Para j=ini hasta i-1
                        Si (y[i] > y[j]) Entonces
                              r[i] ← r[i]+1
                        Fin Si
                  Fin Para
            Fin Para
            Retorno
      Fin Si

      posmedia ← (ini+fin)/2
      puntosS1 ← 0.0

      DyV_2_rec (y, posy, r, ini, posmedia, umbral)
      DyV_2_rec (y, posy, r, posmedia+1, fin, umbral)

      Quicksort (y, posy)

      Para i=ini hasta fin
            Si ((posy[i]>=ini) y (posy[i]<=fin)) Entonces
                  Si (posy[i] > posmedia) Entonces
                        rango[posy[i]] ← rango[posy[i]] +
            puntosS1
                  Si No
                        puntosS1 ← puntosS1+1
                  Fin Si
            Fin Si
      Fin Para

Fin Procedimiento
```

Análisis teórico

Comenzamos analizando la función *dyv_rango_1* que actúa de interfaz para la función recursiva:

Nota: Todos los logaritmos expresados en este apartado son en base 2.

```
(1)   void dyv_rango_2 (const vector < pair<float,float> > S,
      vector<float> &rango, const int umbral)
(2)   {
(3)       unsigned int i;
(4)       const unsigned int size=S.size();
(5)
(6)       vector<float> x(size),y(size), rango_desordenado(size);
(7)       vector<unsigned int> posx(size),posy(size);
(8)
(9)       for (i=0;i<size;i++)
(10)      {
(11)          x[i] = S[i].first;
(12)          posx[i] = i;
(13)          rango_desordenado[i] = 0.0;
(14)      }
(15)
(16)      dyv_rango_qs(x,posx,0,size-1);
(17)
(18)      for (i=0;i<size;i++)
(19)      {
(20)          y[i] = S[posx[i]].second;
(21)          posy[i] = i;
(22)      }
(23)
(24)      dyv_rango_qs(y,posy,0,size-1);
(25)
(26)      dyv_rango_2_rec(y, posy, rango_desordenado, 0, size-1, umbral);
(27)
(28)      rango.resize(size);
(29)      for (i=0;i<size;i++)
(30)      {
(31)          rango[posx[i]] = rango_desordenado[i];
(32)      }
(33)  }
```

Los resultados del análisis teórico de la versión anterior nos sirven para esta versión ya que en la función interfaz sólo cambia la línea 24, sólo tenemos que sumar al tiempo anterior *nlogn* quedando: ***T(n) = 2nlogn + 3n + f_1(n)***.

Ahora sólo queda calcular el tiempo correspondiente a la función recursiva $f_1(n)$.

```
(1)   void dyv_rango_2_rec (const vector<float> &y, const vector<unsigned
      int> &posy, vector<float> &rango, const unsigned int &ini, const
      unsigned int &fin, const unsigned int &umbral)
(2)   {
(3)
(4)
(5)       if (fin-ini+1<=1) return;
(6)
(7)       if (fin-ini+1<=umbral)
(8)       {
(9)           dyv_rango_it(y, rango, ini, fin);
(10)          return;
(11)      }
(12)
(13)      unsigned int posmedia = (ini+fin)/2,i;
(14)      float puntosS1 = 0.0;
(15)
(16)      dyv_rango_2_rec(y, posy, rango, ini, posmedia, umbral);
(17)      dyv_rango_2_rec(y, posy, rango, posmedia+1, fin, umbral);
(18)
(19)      for (i=0;i<y.size();i++)
(20)      {
(21)          if (posy[i] >=ini && posy[i]<=fin)
(22)          {
(23)              if (posy[i] > posmedia)
(24)              {
(25)                  rango[posy[i]]+=puntosS1;
(26)              }
(27)              else puntosS1++;
(28)          }
(29)      }
(30)  }
```

Que sólo cambia con respecto a la versión 1 que no realiza el ordenamiento por y, por lo que sólo tenemos que restarle *nlogn*, quedando ***f_1(n) = 2·n + n·logn – 2***.

Finalmente el resultado es exactamente igual que el anterior:

$$T(n) = n\log n + n^2 + 3n,\ n <= umbral$$
$$T(n) = 3n\log n + 5n - 2,\ \ n > umbral.$$

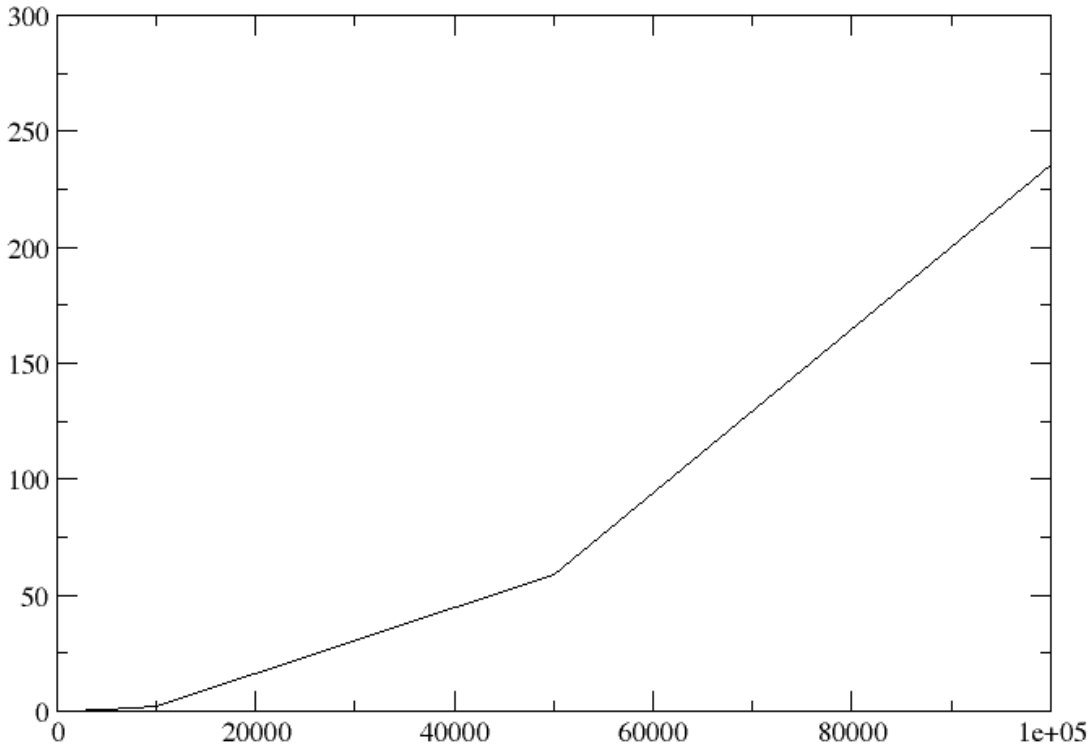
La función tendrá el orden ***O(nlogn) para n>umbral y O(n^2) en caso contrario***.

Análisis empírico

Visto en el apartado *1.1*.

Análisis híbrido

Llevando los datos obtenidos en el apartado anterior obtenemos la siguiente gráfica:



Introduciendo como fórmula la siguiente: y = *a0*x*log2(x) + a1*x + a2* obtengo los siguientes parámetros:

a0 = 9.7917·10^{-3}
a1 = -0.0139693
a2 = 2.71797

Y coeficiente de relación ***0.997265***, muy próximo a *1*.

1.2.3 Cálculo del umbral óptimo

Para este apartado he realizado el siguiente programa:

```
#include <time.h>
#include <stdlib.h>
#include <iostream>
#include <map>
#include <vector>



using namespace std;


void dyv_rango_1 (const vector < pair<float,float> > S, vector<float>
&rango, const int umbral);
void dyv_rango_2 (const vector < pair<float,float> > S, vector<float>
&rango, const int umbral);


void rellena(vector<pair<float, float> > &v, const unsigned int &n)
{
    v.resize(n);
    for (unsigned int i=0;i<n;i++)
    {
        v[i].first = (float) rand()/RAND_MAX;
        v[i].second = (float) rand()/RAND_MAX;
    }
}



int main(int argc, char **argv)
{
    vector<float> rango;
    vector<pair<float, float> > v;

    clock_t antes,despues;
    unsigned int i=0,n=10000, umbral=1, NUM=100;
    double tiempo1=0.0, tiempo2;
    unsigned int fase = 0;
(...)
```

```
    srand(time(NULL));

    do
    {
        tiempo2=0.0;
        for (i=0;i<NUM;i++)
        {
            rellena(v, n);

            antes = clock();
            dyv_rango_1(v, rango, umbral);
            dyv_rango_2(v, rango, umbral);
            despues = clock();

            tiempo2 = tiempo2 + ( (double)(despues-antes) / (double)
(CLOCKS_PER_SEC) );
        }

        tiempo2 = tiempo2 / (2.0*NUM);

        cout<<fase<<','<<umbral<<','<<tiempo2<<endl;

        if (tiempo2<tiempo1)
        {
            if (fase==0)
                umbral = umbral/2;
            else umbral--;

            fase++;
        }
        else if (fase!=2)
                umbral = (fase==0)?(umbral * 2):(umbral+1);

        tiempo1=tiempo2;

    }
    while (fase!=2);

    cout<<"El umbral optimo es:"<<umbral<<endl;
}
```

El cual va buscando el umbral comenzado por *1* y duplicando la cifra por la que vaya, si en un momento dado el tiempo disminuye vuelve al umbral anterior y va aumentando de 1 en 1. Si llega un punto en que el tiempo vuelve a aumentar el umbral anterior era el óptimo.

Tras varias pruebas el umbral óptimo me lo sitúa en torno a ***12***.

1.2.4 Consideraciones adicionales

El módulo que realiza la ordenación mediante el algoritmo *quicksort* se llama ***dyv_rango_qs***, este módulo realiza una ordenación un tanto especial, ya que ordena de forma creciente un vector de flotantes y un segundo vector de posiciones que sigue el mismo orden que el anterior, de forma que más adelante se pueda recuperar el orden original del vector.

Esta función es usada por las versiones 1 y 2 del algoritmo.

Por último, comentar que las funciones recursivas implementadas para cada versión del algoritmo nunca se les pasa como parámetro las coordenadas *x* ya que previamente he ordenado por esa variable por lo que supongo a efectos de todos los cálculos que ya se encuentra ordenado y no es necesaria ninguna comprobación para *x*.

2. Tornillos y tuercas

Descripción en pseudocódigo

Función principal que sirve de interfaz con la recursiva:

```
Procedimiento DyV_tor_y_tuer (tor, tuer, tor_ordenado,
      tuer_ordenado: vector)


      tor_ordenado ← tor
      tuer_ordenado ← tuer


      DyV_tor_y_tuer_rec (tor_ordenado, tuer_ordenado, 0,
      tamaño(tor)-1 )


Fin Procedimiento
```

Función recursiva:

```
Procedimiento DyV_tor_y_tuer_rec (tor, tuer: vector; ini, fin:
      entero)


      Si (fin-ini+1<=1) Entonces
            Retorno
      Fin Si


      pos, i, izq, der, posmedia: enteros
      miTor: tornillo
      miTuer: tuerca


      pos ← Aleatorio()


      Para i=ini hasta fin
            Si (compara(miTor, tuer[i])==0) Entonces
                  miTuer = tuer[i]
                  Intercambia(tuer[i], tuer[ini])
            Fin Si
      Fin Para


      izq ← ini+1
      der ← fin
(...)
```

```
Mientras (izq < der)
      Mientras ((izq < der) y (compara(miTor,tuer[izq]==1))
            izq ← izq+1
      Fin Mientras
      Mientras ((izq < der) y (compara(miTor,tuer[der]==-1))
            der ← der-1
      Fin Mientras

      Intercambia(tuer[izq], tuer[der])
Fin Mientras

Si (compara(miTor,tuer[izq])==1) Entonces
      posmedia ← izq
Si No
      posmedia ← izq-1
Fin Si
Intercambia(tuer[posmedia], tuer[ini])

izq ← ini+1
der ← fin
Mientras (izq < der)
      Mientras ((izq < der) y (compara(tor[izq],miTuer==-1))
            izq ← izq+1
      Fin Mientras
      Mientras ((izq < der) y (compara(tor[der],miTuer==1))
            der ← der-1
      Fin Mientras
      Intercambia(tor[izq], tor[der])
Fin Mientras

Si (compara(tor[izq],miTuer)==-1) Entonces
      posmedia ← izq
Si No
      posmedia ← izq-1
Fin Si
Intercambia(tor[izq], tor[der])

DyV_tor_y_tuer_rec (tor_ordenado, tuer_ordenado, ini,
posmedia-1 )
DyV_tor_y_tuer_rec (tor_ordenado, tuer_ordenado, posmedia+1,
fin )
```

Análisis teórico

La función que hace de interfaz con la recursiva apenas tiene instrucciones salvo la llamada principal a la función recursiva por lo que paso directamente al análisis de esta última:

Nota: Todos los logaritmos expresados en este apartado son en base 2.

```
(1)   void dyv_tornillos_y_tuercas_rec (vector<tornillo> &tor,
(2)                                     vector<tuerca> &tuer,
(3)                                     const unsigned int &ini,
(4)                                     const unsigned int &fin)
(5)   {
(6)       if (fin-ini+1<=1) return;
(7)
(8)       unsigned int i, izq=ini+1,der=fin, posmedia;
(9)
(10)
(11)      unsigned int pos = ini+static_cast<int>
      (static_cast<double>(fin-ini+1)*rand()/(RAND_MAX+1.0));
(12)
(13)      tornillo miTor = tor[pos];
(14)      tuerca miTuer;
(15)      bool salir = false;
(16)
(17)
(18)      if (pos!=ini)
(19)          swap(tor[pos], tor[ini]);
(20)
(21)
(22)      for (i=ini;i<=fin && !salir;i++)
(23)          if (!compara(miTor, tuer[i]))
(24)          {
(25)              miTuer = tuer[i];
(26)              if (i!=ini)
(27)                  swap(tuer[i], tuer[ini]);
(28)              salir = true;
(29)          }
```

```
(30)
(31)        while (izq < der)
(32)        {
(33)            while (izq<der && (compara(miTor,tuer[izq])==1)) izq++;
(34)            while (izq<der && (compara(miTor,tuer[der])==-1)) der--;
(35)
(36)            swap(tuer[izq],tuer[der]);
(37)        }
(38)
(39)        posmedia = (compara(miTor,tuer[izq])==1)?izq:izq-1;
(40)        if(posmedia!=ini) swap(tuer[posmedia], tuer[ini]);
(41)
(42)
(43)        izq = ini+1; der = fin;
(44)        while (izq < der)
(45)        {
(46)            while (izq<der && (compara(tor[izq],miTuer)==-1)) izq++;
(47)            while (izq<der && (compara(tor[der],miTuer)==1)) der--;
(48)
(49)            swap(tor[izq],tor[der]);
(50)        }
(51)
(52)        posmedia = (compara(tor[izq],miTuer)==-1)?izq:izq-1;
(53)        if(posmedia!=ini) swap(tor[posmedia], tor[ini]);
(54)
(55)        dyv_tornillos_y_tuercas_rec(tor, tuer, ini, posmedia-1);
(56)        dyv_tornillos_y_tuercas_rec(tor, tuer, posmedia+1, fin);
(57)    }
```

Comenzamos analizando las llamadas recursivas de las líneas 55 y 56 de las cuales se deducen el tiempo ***T(n) = 2T(n/2-1) + f(n)*** donde *f(n)* es el tiempo del resto de la función.

Obtenemos por la línea 6 que **f(0) = f(1) = 1**.

El tiempo invertido en las instrucciones de la línea 11 hasta la 19 lo podemos despreciar.

De la línea 22 a la 29 tenemos un bucle que se ejecutará en el peor de los casos **n** veces con *ini=0* y *fin=n*, su cuerpo lo tomaremos de orden *O(1)*.

Los bucles de las líneas 31-37 y 44-50 se ejecutarán un máximo de *n/2* veces, por lo que tenemos que *2n/2 =* **n**.

Los tiempos de las líneas 39-43 y 52 y 53 los podemos despreciar.

Finalmente ***f(n) = 2n*** y ***T(n) = 2T(n/2-1) + 2n***

Resolvemos la recurrencia anterior haciendo el cambio de variable $n = 2^m$;

$T(2^m) = 2T(2^{m-1}-1) + 2n$

$T(2^m) – 2T(2^{m-1}-1) = 2n$, *haciendo el cambio* $T(2^m) = t(m)$

$t(m) – 2t(m-1) = 2^{m+1}$, *obtenemos la ecuación de recurrencia*

$x – 2 = 0$
$p(x) = (x – 2)(x - 1)^2$

cuyas raizes son: 2 y 1 (con multiplicidad 2)

la recurrencia queda como $t(m) = c_1 2^m + c_2 1^m + c_3 m 1^m$
calculamos las 3 constantes sabiendo que: t(0) = t(1) = log(1) y t(2) = log(4)

$t(0) = c_1 + c_2 = 0$
$t(1) = 2\cdot c_1 + c_2 + c_3 = 0$
$t(2) = 4\cdot c_1 + c_2 + 2\cdot c_3 = 2$

resolvemos y obtenemos; $c_1 = 2$, $c_2 = -2$ *y* $c_3 = 2$.
sustituimos: $t(m) = 2\cdot 2^m + 2\cdot m - 2$
ahora deshacemos el cambio de variable: $T(n) = 2\cdot n + 2\cdot n\cdot \log n – 2$

Por tanto tenemos: ***T(n) ∈ O(nlogn)***

Análisis empírico

Para este apartado he realizado el siguiente programa:

```
#include <time.h>
#include <stdlib.h>
#include <iostream>
#include <map>
#include <vector>



typedef struct {
   int grosor;
} tornillo;
typedef struct {
   int anchura;
} tuerca;

using namespace std;

void dyv_tornillos_y_tuercas (const vector<tornillo> tor,
                              const vector<tuerca> tuer,
                              vector<tornillo> &tor_ordenado,
                              vector<tuerca> &tuer_ordenado);

void rellena(vector<tornillo> &t, vector<tuerca> &u, const unsigned int
&n)
{
    int pos;

    t.resize(n);
    u.resize(n);

    for (unsigned int i=0;i<n;i++)
    {
        t[i].grosor = -1;
        u[i].anchura = -1;
    }
(...)
```

```
    for (unsigned int i=0;i<n;i++)
    {
        do
        {
            pos = static_cast<int> (static_cast<double>(n)*rand()/
(RAND_MAX+1.0));
        }
        while (t[pos].grosor != -1);

        t[pos].grosor = i;

        do
        {
            pos = static_cast<int> (static_cast<double>(n)*rand()/
(RAND_MAX+1.0));
        }
        while (u[pos].anchura != -1);

        u[pos].anchura = i;
    }
}

int main()
{
    vector<tornillo> t1,t2;
    vector<tuerca> u1,u2;



    clock_t antes,despues;
    unsigned int i,j,n=1,veces;
    const unsigned int NUM_PEQ=100000, NUM_GRANDE=10;
    double tiempo;

    srand(time(NULL));

    for (j=0;j<12;j++)
      {
        veces = ((j>5)?NUM_GRANDE:NUM_PEQ);

        rellena(t1,u1, n);

(...)
```

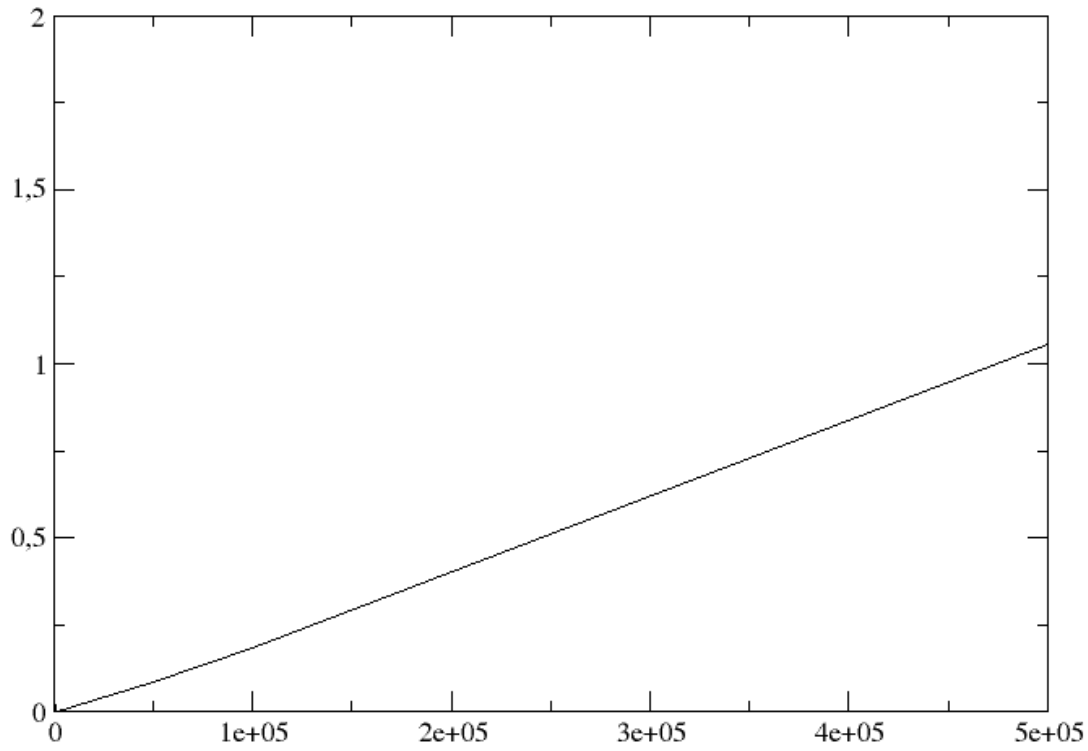
```
        antes = clock ( ) ;
        for ( i =0 ; i<veces; i++)
        {
            dyv_tornillos_y_tuercas(t1,u1,t2,u2);
        }
        despues = clock ( ) ;
        tiempo = ( ( double ) ( despues-antes) / (double)(CLOCKS_PER_SEC
* veces) );

        cout<<n<<' '<<tiempo<<endl;

            n = n * ((j%2)?2:5);
      }

    dyv_tornillos_y_tuercas(t1,u1,t2,u2);
}
```

El cual me hace un incremento de *n* desde 1 hasta 100000, multiplicando n por 2 si j es impar o por 5 si j es par, esto quiere decir que *n* va de la fomar: 1, 5, 10, 50… hasta 100000.

En el fichero correspondiente se puede ver el resultado.

Análisis híbrido

Llevando los datos obtenidos en el apartado anterior obtenemos la siguiente gráfica:



Introduciendo como fórmula la siguiente: y = *a0*x*log2(x) + a1*x + a2* obtengo los siguientes parámetros:

a0 = 1.46163·10^{-7}
a1 = -6.65816·10^{-7}
a2 = 0.00655665

Y coeficiente de relación ***0.999262***, muy próximo a 1.