ALGORÍTMICA

PRÁCTICA 4: BACKTRACKING

Memoria final de la práctica

Ignacio Aguilera Martos
Luis Balderas Ruiz
Diego Asterio de Zaballa Rodríguez
Miguel Ángel Torres López

ugr | Universidad de Granada

CONTENIDOS

1. ORGANIZACIÓN DE LA PRÁCTICA

La práctica 4 trata sobre el desarrollo de algoritmos basados en backtracking que consigan la solución óptima de los problemas propuestos. Usualmente las soluciones obtenidas serán exponenciales, e incluso factoriales, ya que intentaremos resolver problemas NP. En algunos de los algoritmos, la poda nos permitirá conseguir resultados algo más adecuados.

Nos encargamos de resolver el problema 2. La estructura para ello será la siguiente:

- Enunciado del problema
- Resolución teórica (explicación de cada algoritmo)
- Análisis empírico y de la eficiencia híbrida.

En lo que sigue, los miembros del grupo combinamos sistemas operativos y maquinas diferentes para experimentar de la forma más completa y variada la eficiencia de los algoritmos. Estas son las prestaciones de las máquinas:

- Luis: Fujitsu. Intel Core i5. Ubuntu 14.04
- Ignacio: Toshiba. Intel Core i7. Ubuntu 14.04
- Diego: Mac. Intel Core i7. OS X El Capitán
- Miguel Ángel: Toshiba. Intel Core i7. Windows 10

2. PROBLEMA 2: CENA DE GALA

2.1 Enunciado del problema

Se va a celebrar una cena de gala a la que asistirán n invitados. Todos se van a sentar alrededor de una única gran mesa rectangular, de forma que cada invitado tendrá sentados junto a él a otros dos comensales (uno a su izquierda y otro a su derecha). En función de las características de cada invitado (por ejemplo categoría o puesto, lugar de procedencia,...) existen unas normas de protocolo que indican el nivel de conveniencia de que dos invitados se sienten en lugares contiguos (supondremos que dicho nivel es un número entero entre 0 y 100).

El nivel de conveniencia total de una asignación de invitados a su puesto en la mesa es la suma de todos los niveles de conveniencia de cada invitado con cada uno de los dos invitados sentados a su lado. Se desea sentar a los invitados de forma que el nivel de conveniencia global sea lo mayor posible. Diseña e implementa un algoritmo vuelta atrás para resolver este problema.
Realizar un estudio empírico de su eficiencia.

2.2 Solución teórica

Este problema, que puede verse también como el de asignación de tareas y rédito de un rendimiento, sirve de motivación para lo que viene después, la técnica Branch and Bound, ya que pondremos de manifiesto, a lo largo de este desarrollo, que backtracking genera unas soluciones ineficientes de todo punto. De hecho, desde el punto de vista de un algoritmo greedy, se puede asemejar al Problema del Viajante del Comercio, esta vez, con la pretensión de hacer el recorrido más largo posible (justo al contrario que el verdadero TSP). Sin embargo, nos encontramos en la práctica dedicada a backtracking, por lo que nuestros algoritmos irán desde el backtracking más puro, más ineficiente, hasta la introducción de algunas podas, pero siempre en el ámbito de vuelta atrás. Comenzamos con la descripción de los algoritmos:

2.2.1 Algoritmo 1

Estamos ante un problema particular desde el punto de vista de vuelta atrás. En general, cuando se utiliza backtracking es porque no se tiene información suficiente para escoger la rama del árbol donde está la solución óptima. Cuando una rama acaba y no encontramos solución, volvemos atrás hasta encontrar el último nodo antes de tomar la rama sin solución. Sin embargo, en este problema, todas las ramas nos dan una solución, aunque no tiene por qué ser óptima. Ése detalle es determinante en este primer algoritmo, ya que consiste en probar todas las combinaciones posibles de comensales quedándose siempre con la que genera una mayor afinidad hasta el momento. Cuando se recorren todas las posibilidades, el orden de los comensales que tiene una mayor afinidad hasta el momento se convierte en el orden óptimo. Esto da lugar a un algoritmo con eficiencia factorial. Optamos por una implementación recursiva, en la que se tiene una lista de comensales sin colocar aún y el orden generado hasta el momento. La condición de parada de la recursividad se da cuando la lista de comensales sin colocar contiene a una sola persona, por lo que no queda más remedio que colocar a ese último en el sitio que queda.
El código es el siguiente:

```
int suma(list<int> rama, int* afinidades,int n){
	int sum=0;
	list<int>::iterator it;
	list<int>::iterator next_it;
	for(it=rama.begin(); it!=prev(rama.end());++it){
		next_it=it;
		next_it++;
		sum += afinidades[(*it)*n+(*(next_it))];
	}
	sum+= afinidades[(*(rama.begin()))*n+(*(prev(rama.end())))];
	return sum;
}

list<int> backtracking(list<int> rama, int* afinidades, list<int> sinusar,int& maxima_suma, int
n){
		int tam = sinusar.size();
		if(tam <= 1){
			rama.splice(rama.end(),sinusar);
			maxima_suma=suma(rama,afinidades,n);
			return rama;
		}else{
			list<int>::iterator it;
			list<int> res;
			list<int> aux2;
			for(it = sinusar.begin(); it != sinusar.end(); it++){
				int aux = *it;
				it = sinusar.erase(it);
				rama.push_back(aux);
				aux2 = backtracking(rama, afinidades, sinusar,maxima_suma, n);
				if(aux2.size() != 0){
					res = aux2;
				}
				it = sinusar.emplace(it,aux);
				rama.remove(aux);
			}
			return res;
		}
}
```

2.2.2 Algoritmo 2

El algoritmo es un backtracking generado, en lugar de con un árbol, con llamdas recursivas y haciendo uso de la pila. La llamada guarda el mejor tiempo hasta el momento como parámetro por referencia. De este modo, cada vez que encontremos una rama mejor que la mejor que teníamos hasta el momento, solo será necesario cambiar el valor del máximo en una "hoja del arbol". Se ha implementado la poda de nodos para hacer un poco más eficiente el algoritmo. Para ello se tiene en cuenta la eficiencia no se ha podido conseguir en los nodos anteriores y la diferencia entre la mejor rama hasta el momento y la rama actual.

Para evaluar la calidad de una rama se ha usado la función afinidadNoConseguida, que mide la suma de las afinidades que no se han podido conseguir con las personas sentadas hasta el momento.

```
//Cuenta la afinidad que ya has perdido cogiendo la distribución por ahora hecha
en rama
int afinidadNoConseguida(list<int> rama, int* afinidades, int n){
      int res = 0;
      list<int>::iterator itnext ;
      for(list<int>::iterator it = rama.begin(); it != prev(rama.end()); it++){
            itnext = it;
            itnext++;
            res += afinidades[(*it)*n+(*(itnext))];
      }
      res += afinidades[(*(rama.begin()))*n+(*(prev(rama.end())))];
      return 100*rama.size() - res;
}

list<int> backtracking(list<int> rama, int* afinidades, list<int> sinusar, int
&bestafinlost, int n){
      if(afinidadNoConseguida(rama,afinidades,n) >= bestafinlost){
            list<int> res;
            res.clear();
            return res;
      }else{
            int tam = sinusar.size();
            if(tam <= 1){
                  rama.splice(rama.end(),sinusar);
                  bestafinlost=afinidadNoConseguida(rama,afinidades,n);
                  return rama;
            }else{
                  list<int>::iterator it;
                  list<int> res;
                  list<int> aux2;
                  for(it = sinusar.begin(); it != sinusar.end(); it++){//   Crea
una rama por cada persona que no este sentada
                        int aux = *it;
                        it = sinusar.erase(it);
                        rama.push_back(aux);
                        aux2   =   backtracking(rama,   afinidades,   sinusar,
bestafinlost, n);
                        if(aux2.size() != 0){
                              res = aux2;
                        }
                        it = sinusar.emplace(it,aux);
                        rama.remove(aux);
                  }
                  return res;
            }
      }
}
```

ALGORITMOS MIXTOS

Tras proponer las soluciones backtracking más puristas, introducimos ahora unos algoritmos “mixtos” en el que se combina backtracking pero siempre intentando tomar la mejor de las soluciones y así ahorrar costes computacionales, de memoria y tiempo. Son los siguientes:

2.2.3 Algoritmo 3

El algoritmo consiste en coger en orden los comensales y comprobar todas las posibles colocaciones del mismo en la mesa obteniendo para cada comensal la mejor colocación del mismo en cada momento. Visto en cada momento lo que el algoritmo hace es hallar soluciones parciales al problema original tomando siempre la mejor. La manera de saber que dicha posición es la mejor para el comensal es hallar la suma de las afinidades que tiene con su compañero a la derecha e izquierda. Si en algún momento obtiene una suma de afinidades igual a 200 habremos conseguido el máximo y por tanto el algoritmo no continuará comprobando colocaciones para el comensal actual y pasará al siguiente.
El código es el siguiente:

```
void ColocaComensales(short int** matriz, int n_comensales, list<int>&
resultado_final)
{
   //Recorro la lista de resultados para ver el mejor lugar para insertar el
siguiente comensal.
   //El índice i es el comensal actual y el j es el que recorre la lista
resultado_final.
  for(int i = 0; i < n_comensales; ++i)
  {
    list<int>::iterator mejor_posicion = resultado_final.begin();
    int mejor_afinidad = 0;

          for(list<int>::iterator iter = resultado_final.begin(); iter !=
resultado_final.end(); ++iter)
    {

      //Hallo el iterador de antes de end.
      list<int>::iterator antes_end = resultado_final.begin();
            for(list<int>::iterator iter2 = resultado_final.begin(); iter2!
=resultado_final.end(); ++iter2)
      {
        list<int>::iterator aux = iter2;
        ++aux;
        if(aux!=resultado_final.end())
          antes_end = aux;
      }

       //Asigno los pesos de cada comensal para sus acompañantes a izquierdas y
derechas.
        //La lista la considero circular, el primero esta al lado del último y
viceversa.
      int afinidad_izq = 0, afinidad_der = 0;
      if(iter == resultado_final.begin() || iter == antes_end)
      {
        list<int>::iterator aux = iter;
        ++aux;
        afinidad_izq = matriz[i][*iter];
          afinidad_der = (aux!=resultado_final.end())?matriz[i][*aux]:matriz[i]
[resultado_final.front()];
      }
      else
      {
        list<int>::iterator aux = iter;
        ++aux;
        afinidad_izq = matriz[i][*iter];
        afinidad_der = matriz[i][*aux];
      }
```

```
      //Cambio mejor afinidad y mejor posicion si he encontrado un lugar mejor
donde insertar;
      int puntuacion = afinidad_der + afinidad_izq;
      if(puntuacion>mejor_afinidad)
      {
        mejor_posicion = iter;
        mejor_afinidad = puntuacion;
      }
      if(puntuacion==200)
        break;
    }

    resultado_final.insert(mejor_posicion,i);
  }
}
```

2.2.4 Algoritmo 4

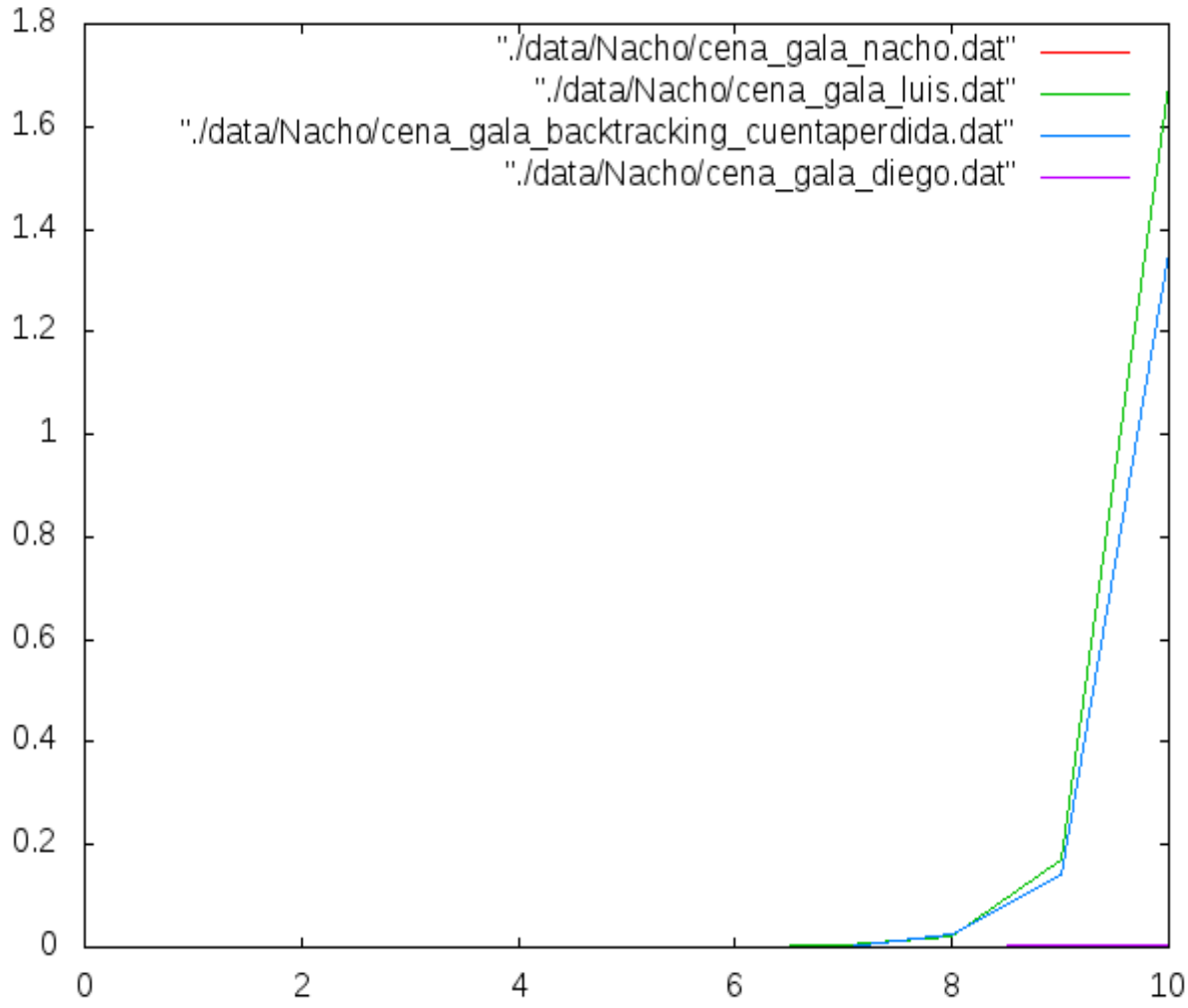
```
class Invitados{
private:
      std::vector<std::vector<int> > preferencia;

      typedef std::priority_queue<std::vector<int> ,std::vector<std::vector<int>
>, std::function<bool(std::vector<int>&, std::vector<int>&)> > mypq_type;

      /**
      * @brief Método para encontrar la afinidad de un vector personas sentadas
en la mesa.
      * @return El valor de la afinidad
      */
  int afinidad(std::vector<int> pers) {

      int afinidad=0, N = pers.size();

      for (int i = 1; i < N; ++i){
            afinidad += preferencia[pers[i-1]][pers[i]];
      }
    afinidad += preferencia[pers[0]][pers[N-1]];

      std::vector<int> noSentadas = noEstan(pers);

      for (size_t i = 0; i < noSentadas.size(); i++) {
            afinidad += preferencia[pers[N-1]][noSentadas[i]];
      }
```

```
      afinidad %= preferencia.size() + 1;

      return afinidad;
  }


      /**
      * @brief Metodo que devuelve los elementos del diccionario que no estan en
el vector
      * @arg parcial vector que se le pasa para encontrar las personas
restantes.
      * @return Los elementos que no estan en parcial y si estan en el map
      */
      std::vector<int> noEstan(std::vector<int> parcial) const{
        std::vector<int> solucion;

        for (size_t i = 0; i < preferencia.size(); ++i){
              if (std::find(parcial.begin(),parcial.end(),i) == parcial.end()){
                    solucion.push_back(i);
              }
        }
        return solucion;
  }
      /**
      * @brief Metodo que encuentra una mesa con buena preferencia con Branch n
Bound
      * @return El vector de personas que la ocupan con el orden adecuado.
      */
  std::vector<int> encuentraMejorMesa() {
    int N = preferencia.size();

        std::function<bool(std::vector<int>&, std::vector<int>&)> comp = [this]
(std::vector<int>& a, std::vector<int>& b) -> bool {return this->afinidad(a) <
this->afinidad(b);};
        mypq_type prior(comp);

        std::vector<int> aux(2);
        aux[0] = 0;
        for (int i = 1 ; i<N ; ++i){
              aux[1] = i;
              prior.push(aux);
        }

        return bestChoice(prior);
}

      /**
      * @brief herramienta recursiva para encontrar la mejor mesa
      * @return en última instancia devuelve la mejor eleccion de mesa
      */
  std::vector<int> bestChoice(mypq_type prior){
         if (prior.top().size() == preferencia.size()){
              return prior.top();
        }
        else{
              std::vector<int> mejores = prior.top();
              prior.pop();
              std::vector<int> nuevosElementos = noEstan(mejores);

              for (auto i : nuevosElementos) {
                    std::vector<int> aux (mejores);
                    aux.push_back(i);
```

```
                    prior.push(aux);
                }
                return bestChoice(prior);
        }
  }

  void generateRandomMatrix(std::vector<std::vector<int> >& m, const int& s){
      std::default_random_engine generator(s);
      std::uniform_int_distribution<int> distribution(0,100);
      auto rng = std::bind(distribution, generator);
      for (size_t i = 0; i < m.size(); i++)
            for (size_t j = i; j < m[i].size(); j++){
                  if (i==j)
                        m[i][j] = 0;
                  else
                        m[j][i] = m[i][j] = rng();
            }
  }

public:
      /**
      * @brief constructor con un vector de nombres
      *     @arg names son los nombres de las personas que habrá que sentar
      */
      Invitados(const int& size){

    preferencia = std::vector<std::vector<int> > (size,std::vector<int>(size));

    generateRandomMatrix(preferencia, size);
  }

      /**
      *     @brief devuelve la puntuacion de la mesa
      * @return un entero que simboliza la puntuacion
      */
      int getMejorAfinidad(){
    return afinidad(encuentraMejorMesa());
  }
};
```

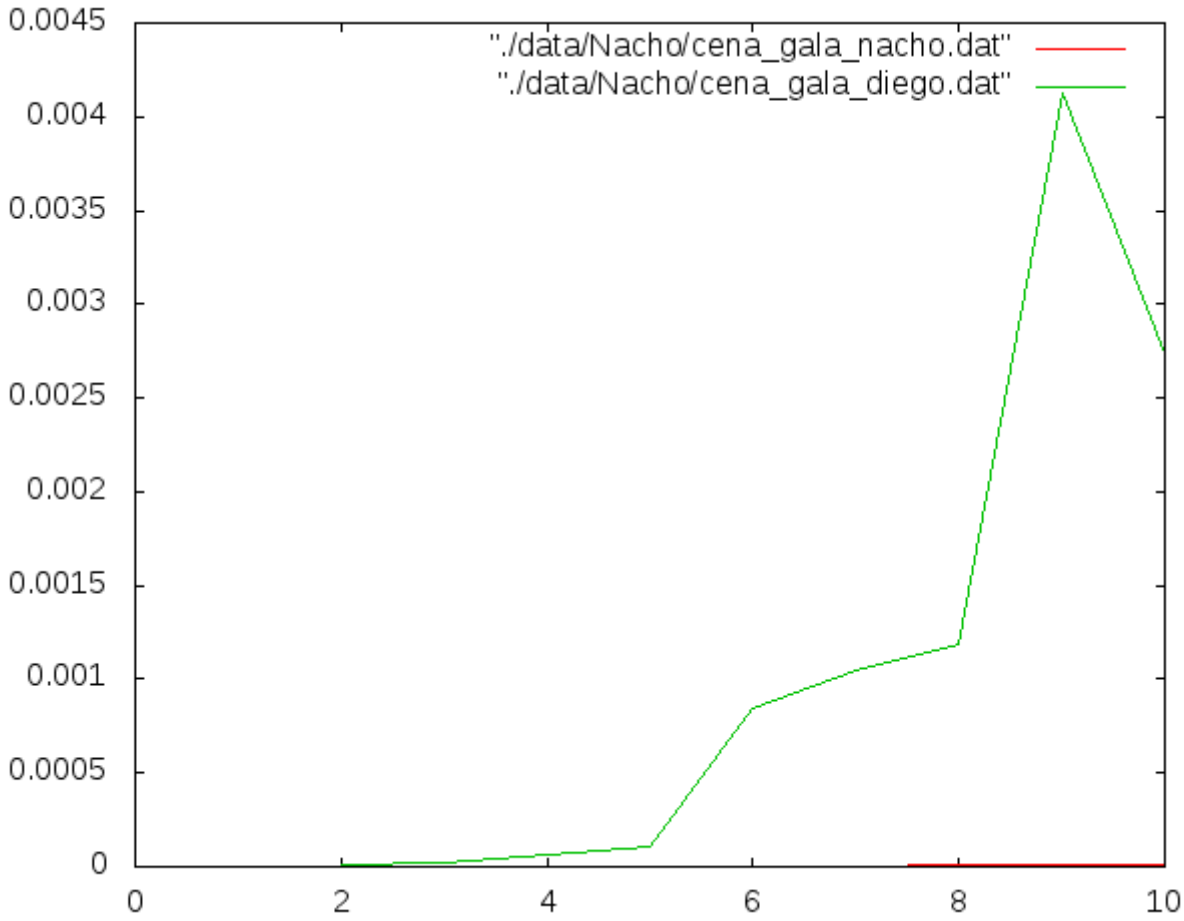
3. ANÁLISIS EMPÍRICO

En la siguiente imagen se ve notoriamente lo que de forma teórica se ha presentado, esto es, la ineficiencia de backtracking en este problema.



Vemos que en los algoritmos vuelta atrás puros el tiempo se dispara, mientras que los mixtos se mantienen muy cercanos a cero. Veremos, más adelante, que la escabilidad también es un factor diferencial, dado que en los backtracking puros apenas pasan de 15 elementos, mientras que los mixtos andan por 250. Además, en aquel backtracking en el que realizamos cierta poda (algoritmo 2) se gana algo de tiempo, mientras que el backtracking más absoluto es el más lento.

Comparamos ahora los backtracking mixto de forma que se vean mejor:



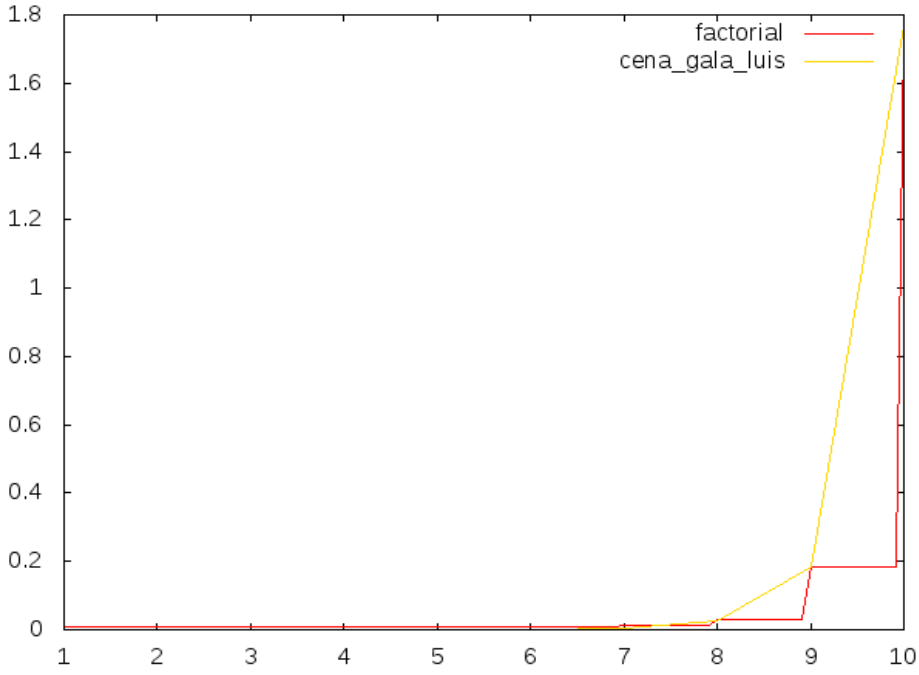
De nuevo se ve que el algoritmo más eficiente es el backtracking del algoritmo 3.

En lo que sigue, se muestran los datos y las gráficas en cada máquina:

FUJITSU

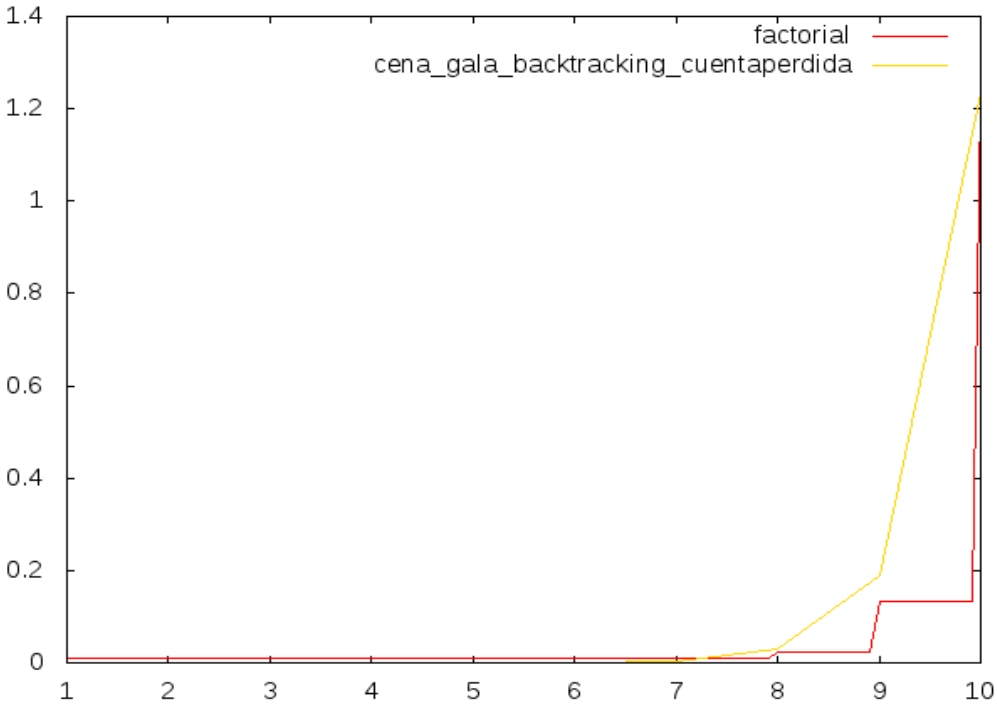
ALGORITMO 1

1	2.703e-06
2	2.775e-06
3	7.477e-06
4	2.4249e-05
5	0.000104139
6	0.000567749
7	0.00319439
8	0.0238218
9	0.18186
10	1.77189



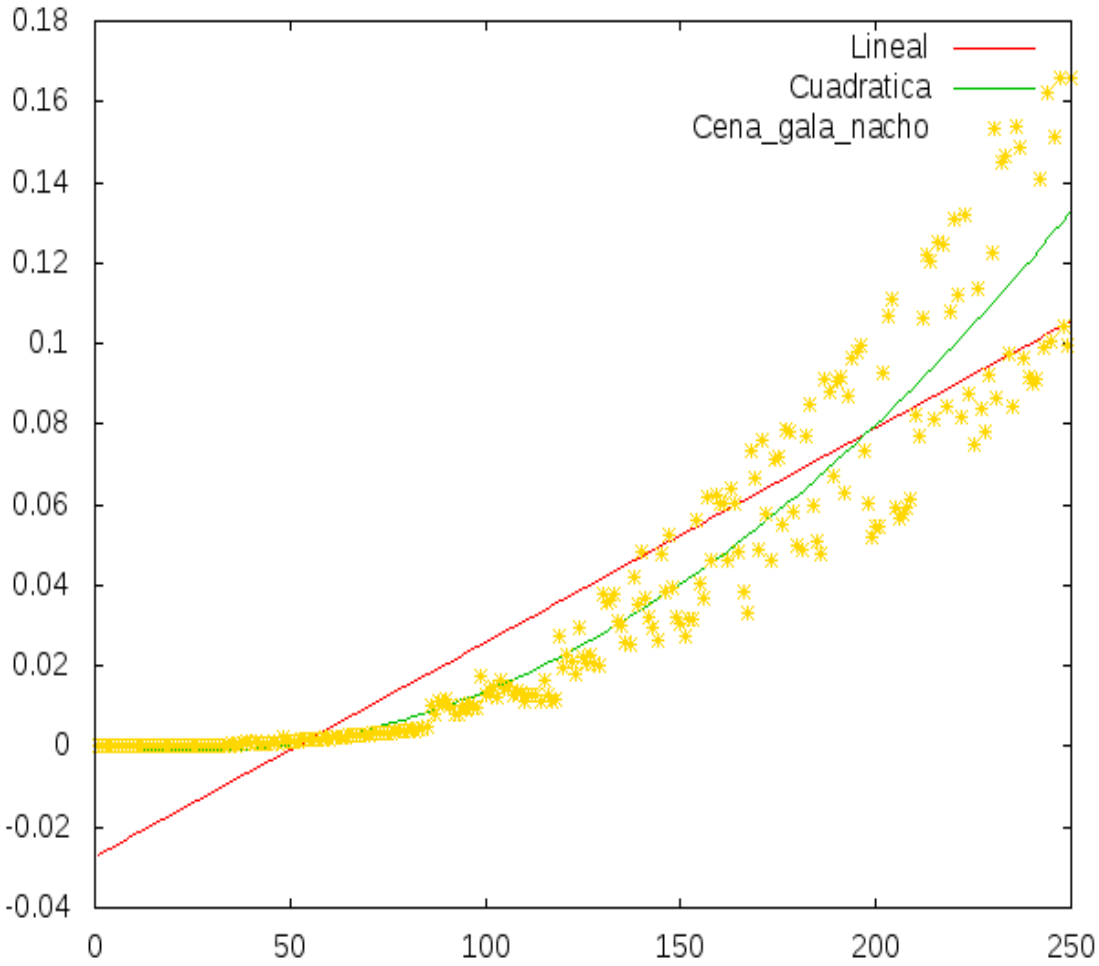
ALGORITMO 2

1	3.621e-06
2	2.907e-06
3	8.498e-06
4	3.8594e-05
5	0.000230846
6	0.000459761
7	0.00375392
8	0.0307815
9	0.189204
10	1.23364



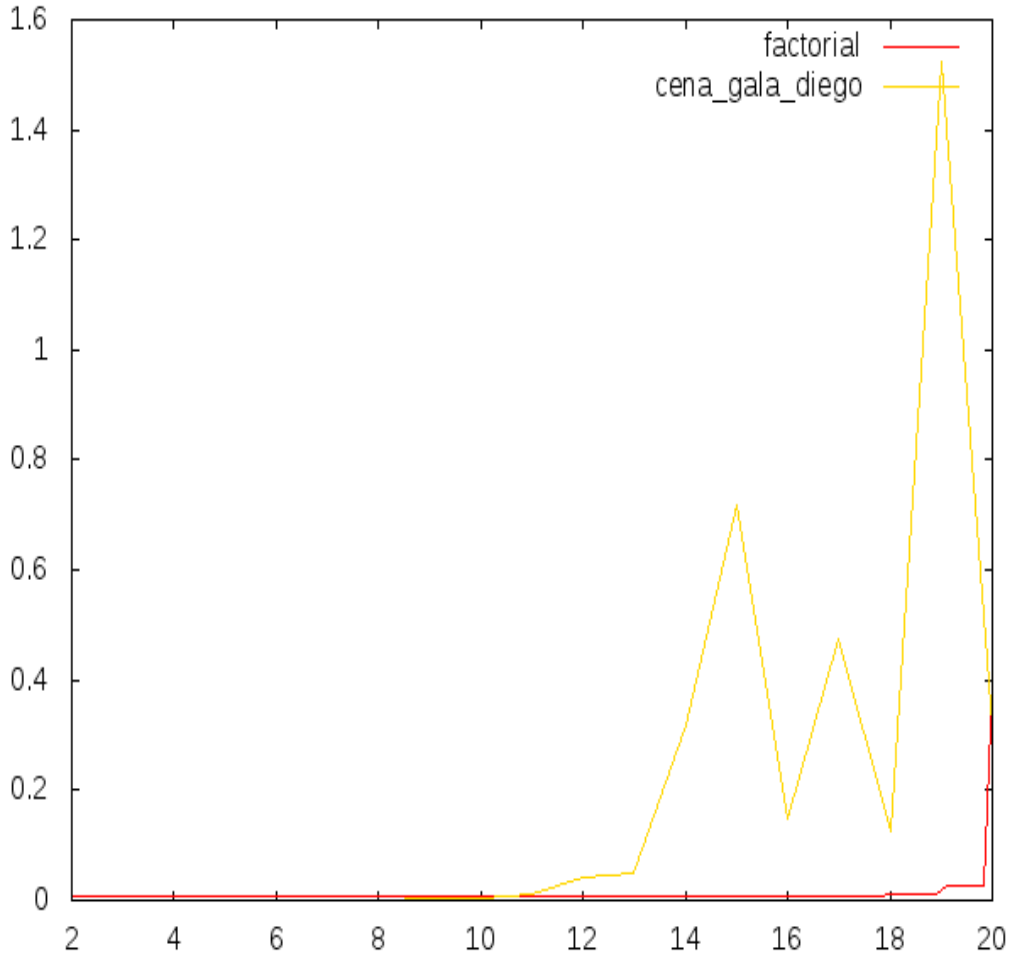
ALGORITMO 3

1 2.626e-06
2 3.356e-06
3 4.435e-06
4 4.353e-06
5 6.366e-06
6 7.679e-06
7 9.364e-06
8 1.1551e-05
9 1.5586e-05
10 1.9644e-05
.
.
.
21 0.000122655
22 0.00012831
.
.
100 0.0125291
101 0.0135794
.
245 0.100327
246 0.151353
247 0.16586
248 0.104468
249 0.0997039
250 0.165674



ALGORITMO 4

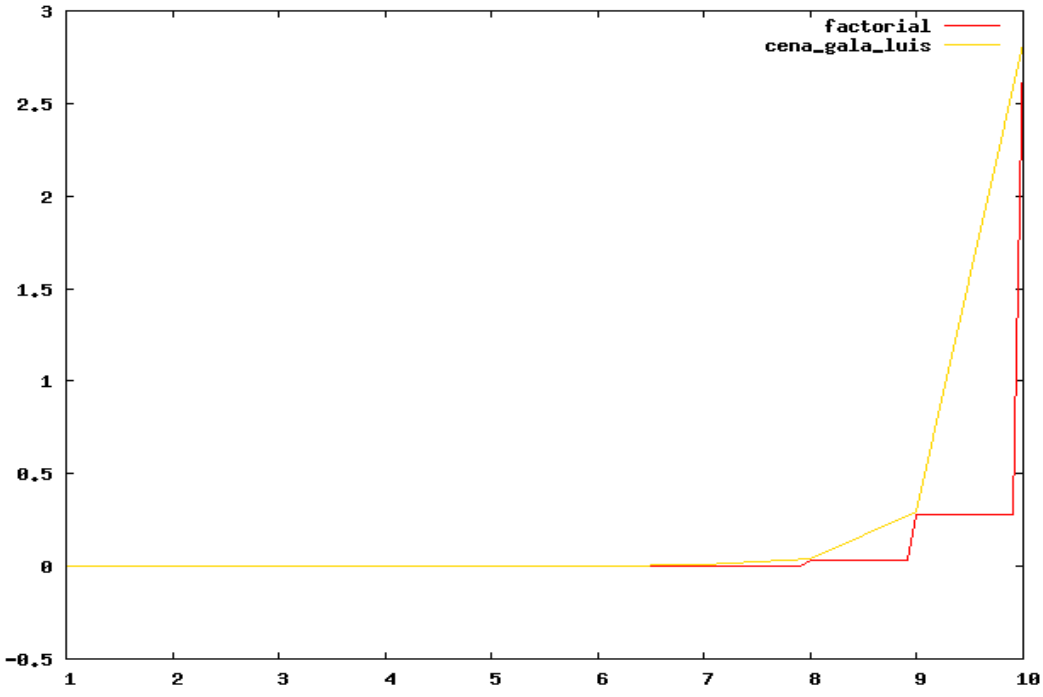
2 9.319e-06
3 2.0928e-05
4 0.000149169
5 0.000151622
6 0.00167596
7 0.00110669
8 0.00165884
9 0.00495385
10 0.00375484
11 0.0115995
12 0.0434867
13 0.050096
14 0.318429
15 0.718604
16 0.148631
17 0.474642
18 0.12725
19 1.5244
20 0.315656



MAC

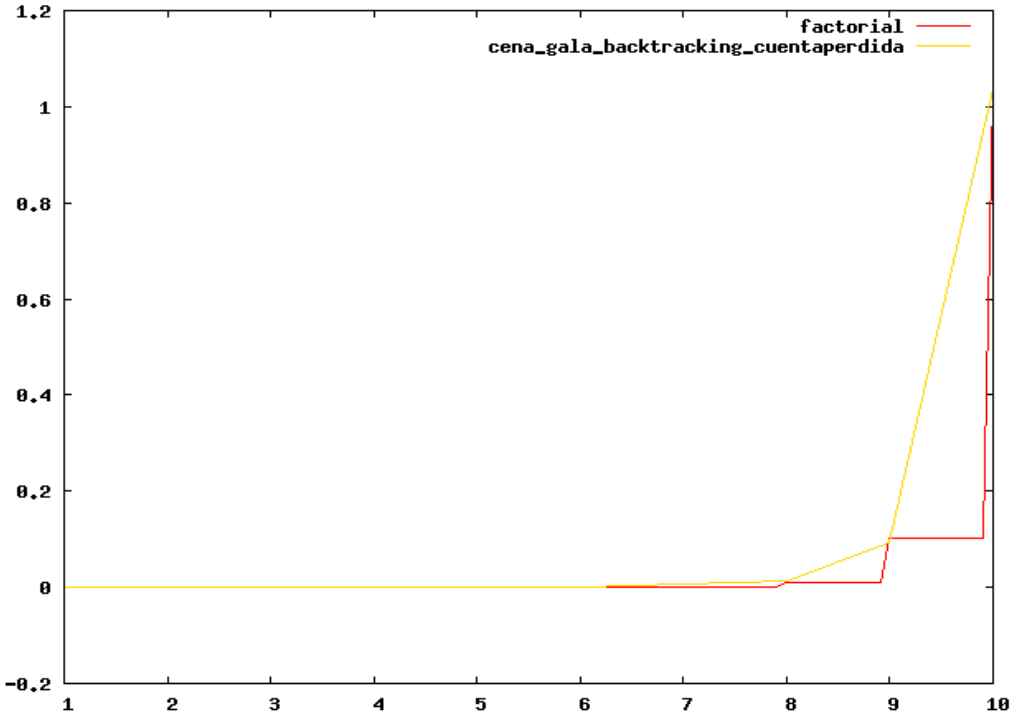
ALGORITMO 1

1 5.156e-06
2 4.823e-06
3 1.3695e-05
4 2.8901e-05
5 0.000118518
6 0.000652574
7 0.00515698
8 0.0397719
9 0.294831
10 2.82424



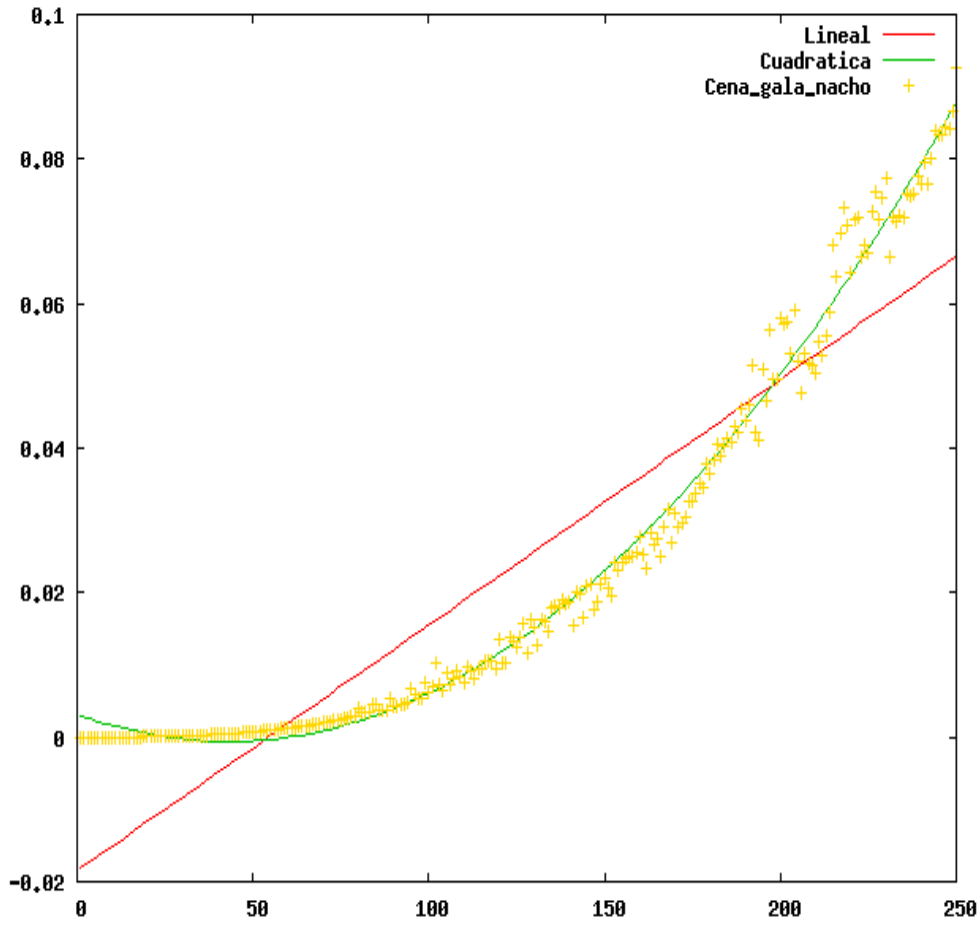
ALGORITMO 2

1 5.62e-06
2 5.226e-06
3 1.4487e-05
4 3.6719e-05
5 8.428e-05
6 0.000478485
7 0.0057068
8 0.0129116
9 0.0922949
10 1.03912



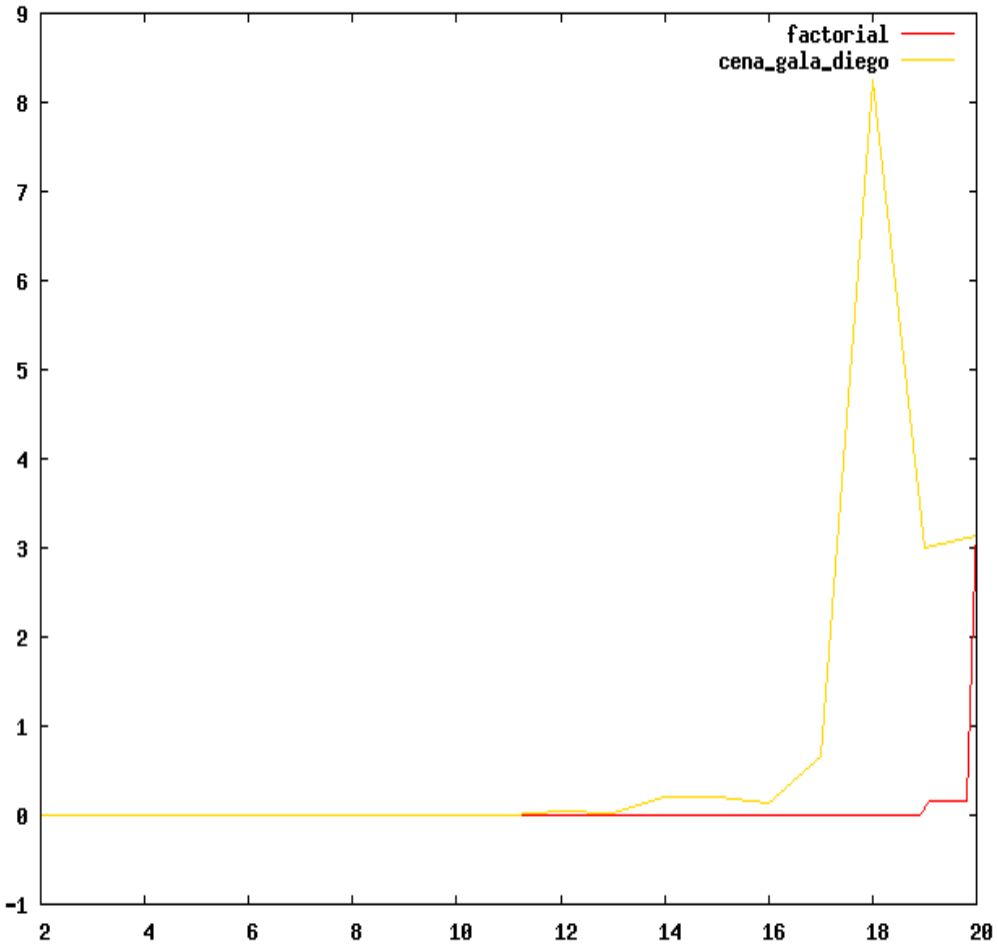
ALGORITMO 3

1 1.458e-06
2 2.212e-06
3 2.625e-06
4 2.441e-06
.
.
80 0.00399585
81 0.00340279
82 0.00335287
.
225 0.0669061
.
241 0.0794097
242 0.0765304
243 0.0801137
244 0.0837826
245 0.0832751
246 0.0832366
247 0.0844254
248 0.0842584
249 0.0865886
250 0.092747



ALGORITMO 4

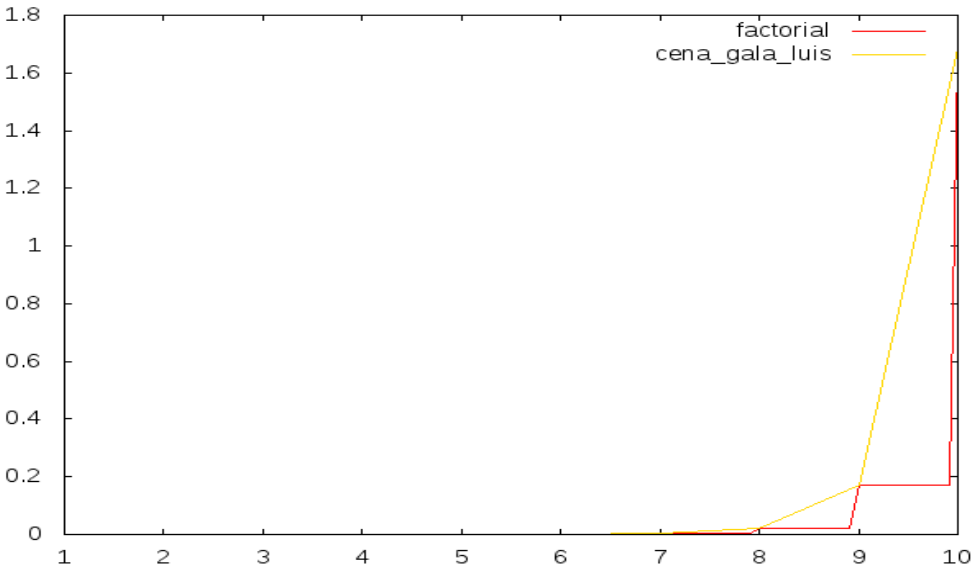
2 0.000494968
3 2.4085e-05
4 8.0132e-05
5 0.000244936
6 0.00106383
7 0.00393598
8 0.0033202
9 0.0028627
10 0.011191
11 0.00709083
12 0.0528241
13 0.0275789
14 0.213566
15 0.21463
16 0.125925
17 0.64898
18 8.24175
19 2.99178
20 3.12603



TOSHIBA (I)

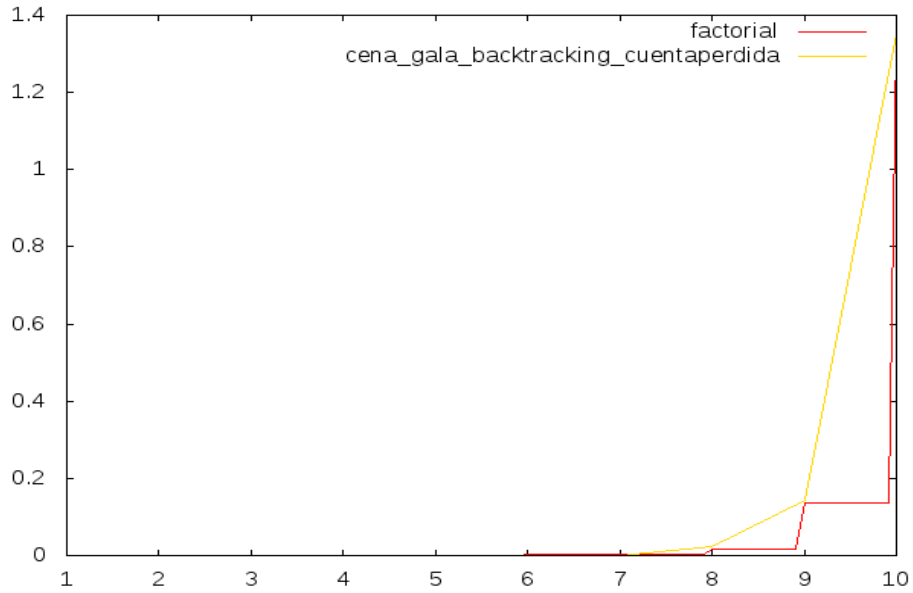
ALGORITMO 1

1 3.238e-06
2 2.305e-06
3 6.519e-06
4 2.929e-05
5 0.000102319
6 0.000435077
7 0.00244438
8 0.0205837
9 0.169754
10 1.68421



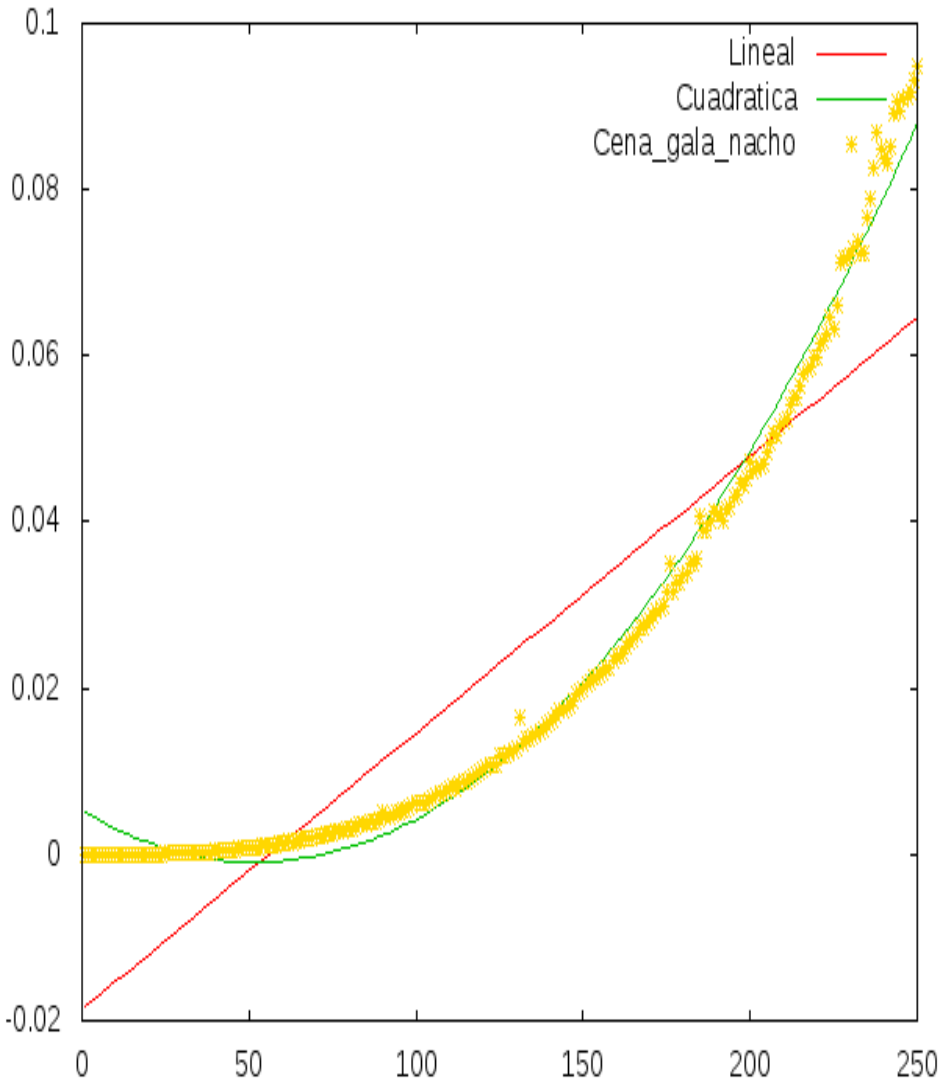
ALGORITMO 2

1 1.623e-06
2 2.76e-06
3 1.032e-05
4 2.5024e-05
5 0.00010206
6 0.000470817
7 0.00102141
8 0.0237846
9 0.14221
10 1.35433



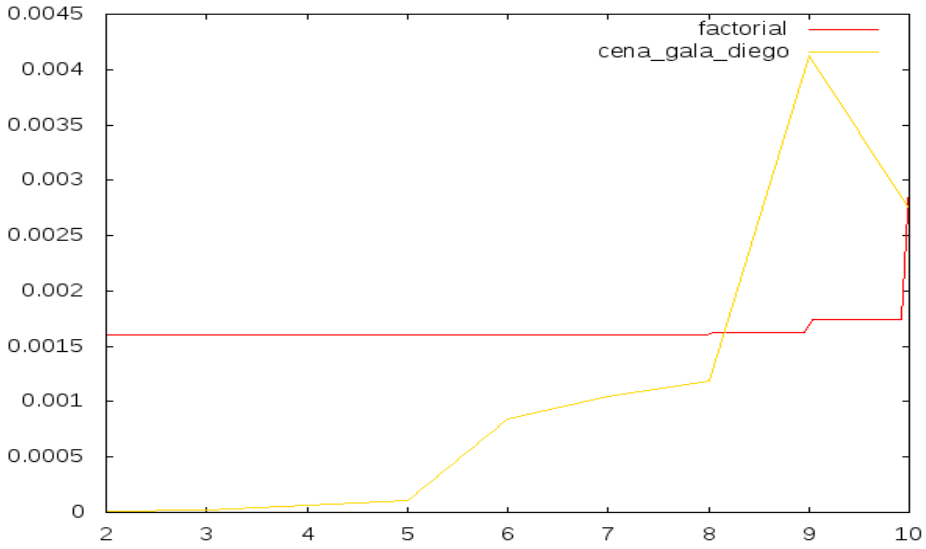
ALGORITMO 3

1 1.737e-06
2 1.595e-06
3 1.857e-06
4 2.847e-06
5 3.191e-06
6 4.434e-06
.
.
100 0.00622688
101 0.00618506
102 0.00618373
103 0.00631792
104 0.00649989
.
174 0.0298704
175 0.0315885
.
220 0.0598653
221 0.0615545
.
241 0.0830422
242 0.0852146
.
248 0.0916732
249 0.093271
250 0.0949944



ALGORITMO 4

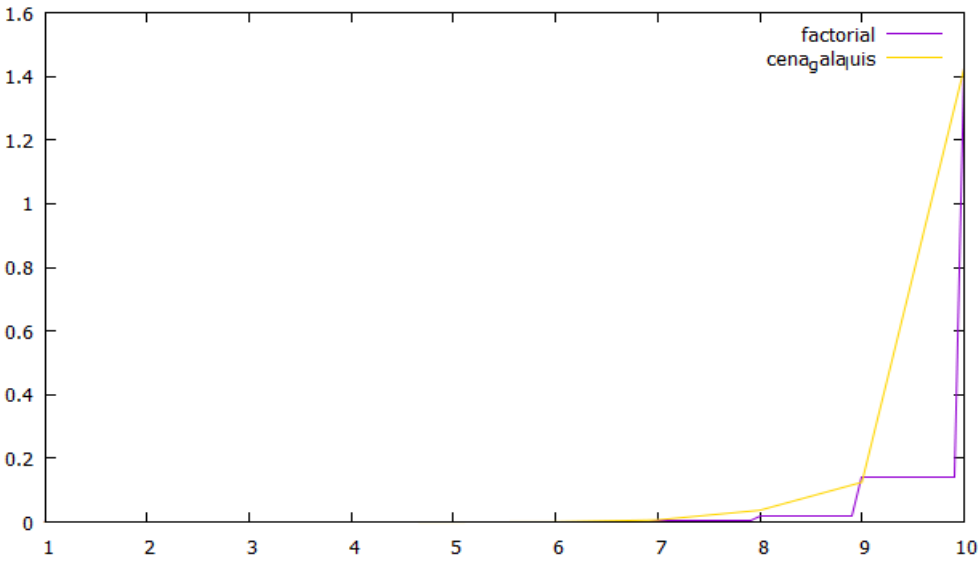
2 7.597e-06
3 1.8612e-05
4 6.4985e-05
5 0.000104315
6 0.000846274
7 0.00105014
8 0.00118681
9 0.00412525
10 0.00273474



TOSHIBA (II)

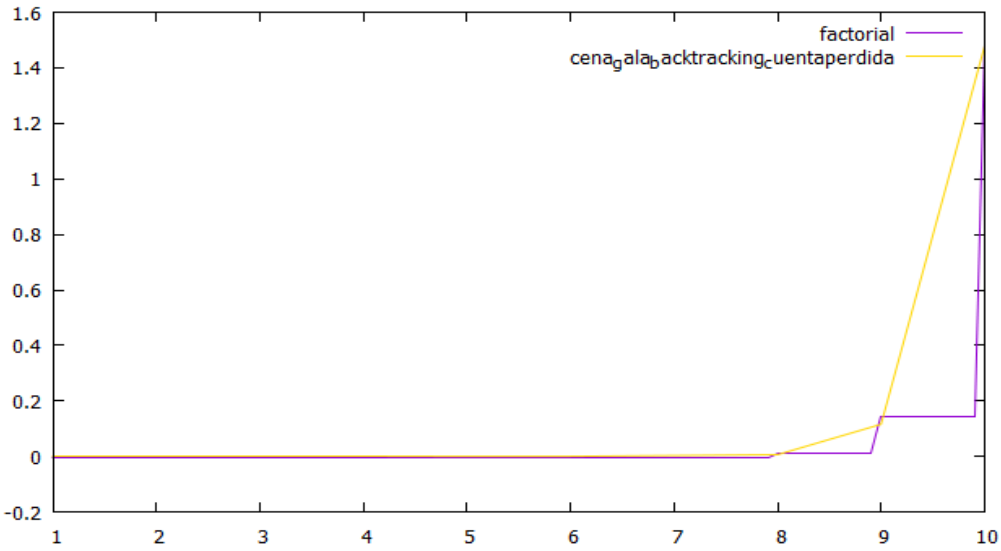
ALGORITMO 1

1 8.12542e-006
2 6.84246e-006
3 2.73698e-005
4 4.87525e-005
5 0.00018945
6 0.00102337
7 0.00684588
8 0.0369937
9 0.125047
10 1.42412



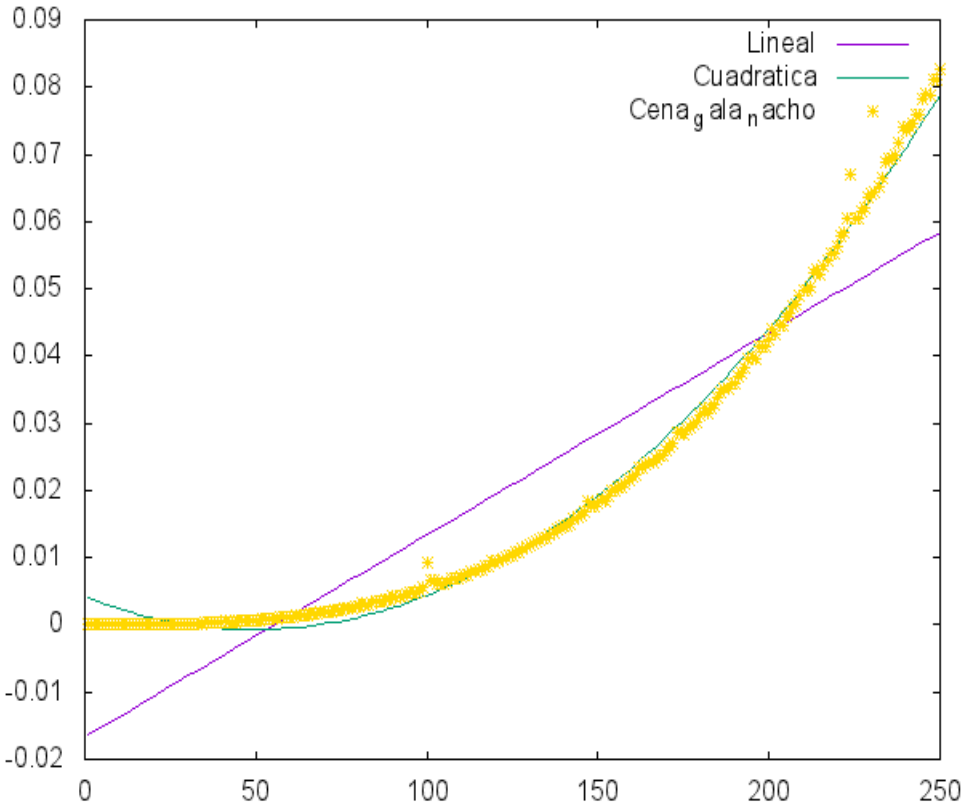
ALGORITMO 2

1 8.55307e-006
2 1.32573e-005
3 2.56592e-005
4 4.36207e-005
5 0.000168923
6 0.000661152
7 0.00331218
8 0.00623219
9 0.115942
10 1.47772



ALGORITMO 3

1 2.13827e-006
2 4.27653e-006
3 3.84888e-006
4 2.13827e-006
5 3.42123e-006
.
.
100 0.00935107
101 0.00660126
102 0.00665557
103 0.00601879
.
.
150 0.0181483
151 0.0185559
152 0.0185212
.
200 0.0424532
201 0.0439132
202 0.0432268
203 0.0445662
204 0.0446611
205 0.0458958
.
.
243 0.0759645
244 0.0757857
245 0.0781199
246 0.0789311
247 0.0786472
248 0.0812439
249 0.0812332
250 0.0826273



ALGORITMO 4

4. BIBLIOGRAFÍA

-Jose Luis Verdegay, “Curso de Teoría de Algoritmos”
-Brassard, Bradley, “Fundamentos de Algoritmia”
-www.gnuplot.com
-Knuth, “The art of computer programming”