UNIVERSIDAD DE GRANADA

Serie unimodal de números

Algorítmica

Aarón Bueno Rodríguez
Bryan Moreno Picamán
Miguel Ángel Rodríguez Serrano

2º Grado Ingeniería Informática

Algoritmo Sencillo

Algoritmo sencillo

El algoritmo que a cualquier persona se le ocurriría para resolver este problema hubiera funcionado de la siguiente manera:

- Avanzamos desde la primera posición del vector de una en una hasta que encontremos que el valor de la posición actual es menor que el de la anterior.
- Devolvemos la posición anterior, que es el índice.

La implementación del algoritmo resultante es la siguiente:

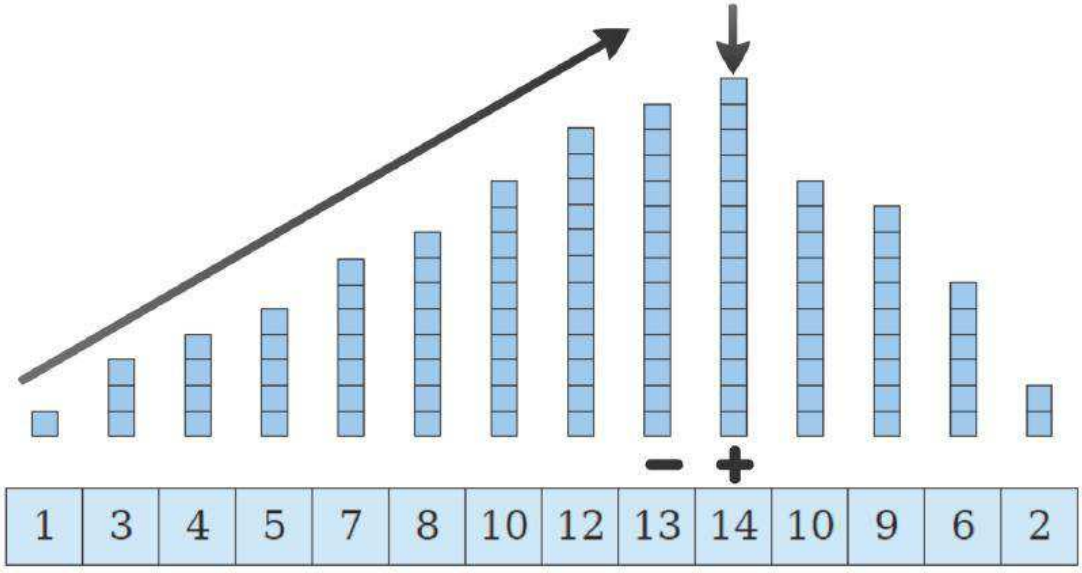
```
int Unimodal(int v[], int ini, int fin){
  int max = ini;
  for (int i = ini + 1; i <= fin; i++){
    if ( v[i] > v[max]){//Todavía no hemos llegado al pico
      max = i;//Guardamos la posicion
    }
    else{//Ya ha encontrado el pico, no hace falta seguir
      return max;//Devolvemos la posición anterior
    }
  }
}
```

Vamos a analizar su eficiencia:

```
int Unimodal(int v[], int ini, int fin){
  int max = ini;                                                    //O(1)
  for (int i = ini + 1; i <= fin; i++){                             //O(n)
    if ( v[i] > v[max]){//Todavía no hemos llegado al pico          //O(1)
      max = i;//Guardamos la posicion                               //O(1)
    }
    else{//Ya ha encontrado el pico, no hace falta seguir
      return max;//Devolvemos la posición anterior                  //O(1)
    }
  }
}
```

Cálculo de Eficiencia Teórica

El algoritmo básico realizaría un *barrido* de todo el vector, hasta encontrar el mayor elemento de todos. Del número de éstos dependería de manera lineal el tiempo que tardase el algoritmo en encontrar el mayor de todos, por ello podemos afirmar que el orden de eficiencia es $O(n)$



Cálculo de Eficiencia Empírica

Con el fin de obtener el tiempo que tarda el algoritmo en ejecutarse, hacemos uso de la librería *ctime*.
Para calcular la eficiencia empírica nos valemos del siguiente script:

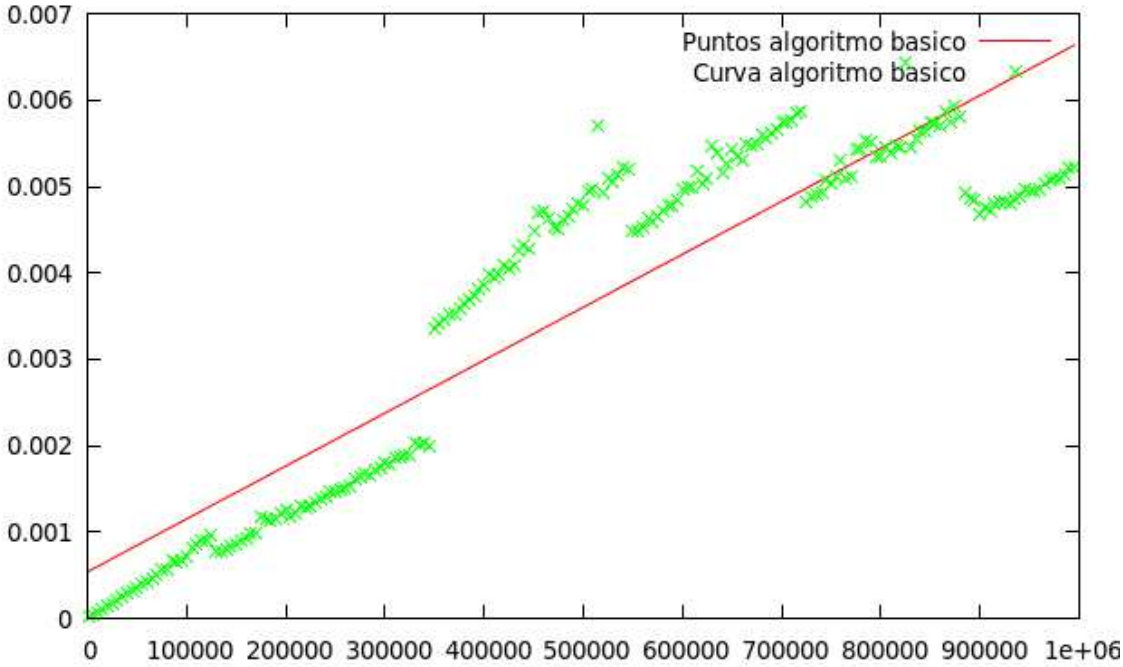
```
1 #!/bin/csh -vx
2 @ i = 100
3 while ( $i < 20000000 )
4 ./algoritmoBasico $i >> basico.dat
5 @ i += 100000
6 end
```

Cálculo de Eficiencia Híbrida

Hemos usado la librería *ctime* para calcular el tiempo. Ahora calculamos su eficiencia híbrida con *gnuplot*. Hemos utilizado la función $f(x) = a_0 \cdot x + a_1$ Las constantes que han salido son:

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a0              = 6.12614e-09      +/- 2.019e-10    (3.296%)
a1              = 0.000539472      +/- 0.0001161    (21.53%)
```

La gráfica resultante tanto de la eficiencia empírica como híbrida ha sido la siguiente:



Los puntos pueden estar dispersos, dependiendo de cuán lejos se encuentre el índice del inicio del vector. Vamos a intentar mejorar este algoritmo creando un algoritmo de tipo Divide y Vencerás.

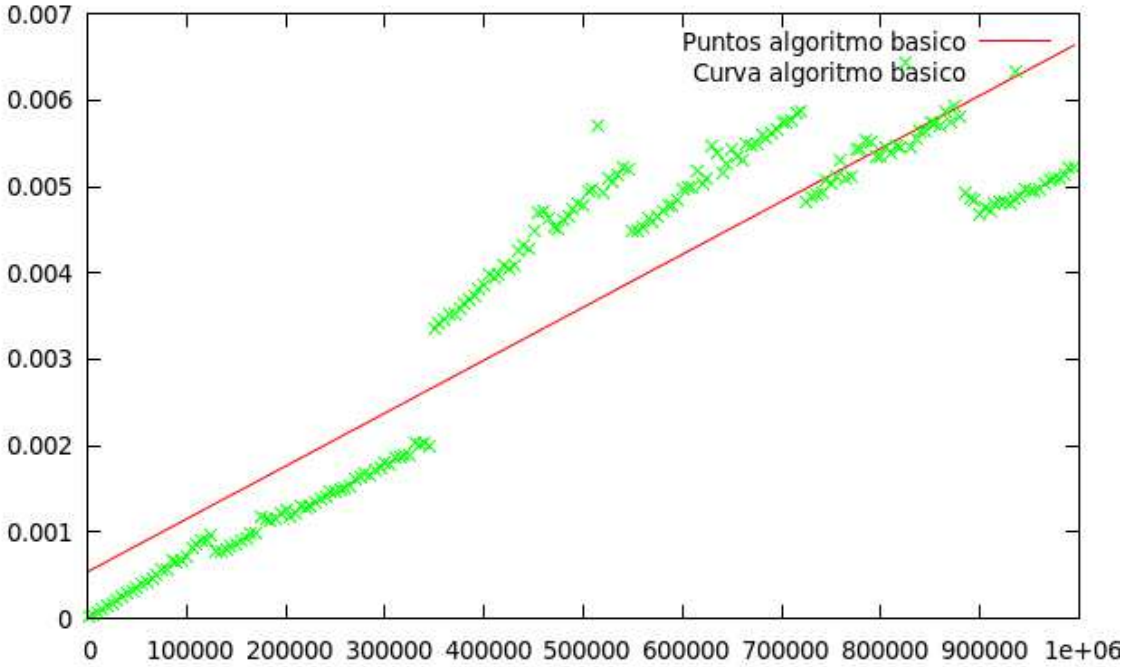
Algoritmo Divide y Vencerás

Cálculo de Eficiencia Híbrida

Hemos usado la librería *ctime* para calcular el tiempo. Ahora calculamos su eficiencia híbrida con *gnuplot*. Hemos utilizado la función $f(x) = a_0 \cdot x + a_1$ Las constantes que han salido son:

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a0              = 6.12614e-09      +/- 2.019e-10    (3.296%)
a1              = 0.000539472      +/- 0.0001161    (21.53%)
```

La gráfica resultante tanto de la eficiencia empírica como híbrida ha sido la siguiente:



Los puntos pueden estar dispersos, dependiendo de cuán lejos se encuentre el índice del inicio del vector. Vamos a intentar mejorar este algoritmo creando un algoritmo de tipo Divide y Vencerás.

Algoritmo Divide y Vencerás

Nuestra idea ha sido la siguiente:

- Nos posicionamos en la mitad del vector.
- Comparamos los valores de las posiciones a ambos lados de la mitad:
  - Si el valor en la posición medio es menor que el que hay en medio-1, el índice se encuentra a la izquierda de medio, así que dividimos en dos el vector entre la posición inicial y medio, y devolvemos la posición del valor máximo que haya entre esas dos partes.
  - Si no, si el valor en la posición medio es menor al de medio+1, se hace lo mismo pero esta vez con la mitad entre medio y la posición final dada.
  - Si no es ninguno de los dos casos anteriores, significa que en realidad el punto medio es el que tiene mayor valor, por lo tanto es el índice y lo devuelve.

La implementación del algoritmo explicado queda de la siguiente forma:

```
int posMax(vector<int> &v,int pos1, int pos2){
  if(v[pos1]>v[pos2])
    return pos1;
  else
    return pos2;
}

int Unimodal(vector<int> &v, int ini, int fin){
  if (ini == fin){//Caso base 1. Sólo tenemos 1 elemento
    return ini;
  }
  else if (ini+1 == fin){//Caso base 2. Tenemos 2 elementos
    return posMax(v,ini,fin);
  }
  else{//Tenemos más de dos elementos
    int medio = (fin+ini)/2, cuarto,primera_pos,segunda_pos, maximo;

    if(v[medio]<v[medio-1]){//El pico está a la izquierda de medio
      cuarto = (medio+ini)/2;//Si tenemos 3 elementos, cuarto == ini
      primera_pos = Unimodal(v,ini,cuarto);
      segunda_pos = Unimodal(v,cuarto+1,medio);
    }
    else if(v[medio]<v[medio+1]){//El pico está a la derecha de medio
      cuarto = (fin+medio)/2;//Si tenemos 3 elementos, cuarto == medio
      primera_pos = Unimodal(v,medio,cuarto);
      segunda_pos = Unimodal(v,cuarto+1,fin);
    }
    else//Caso base 3.El pico es medio
      return medio;

    maximo = posMax(v,primera_pos,segunda_pos);

    return maximo;
  }
}
```

Procedemos a analizar su eficiencia teórica:

```
int posMax(vector<int> &v,int pos1, int pos2){
  if(v[pos1]>v[pos2])                                                    //O(1)
    return pos1;                                                         //O(1)
  else
    return pos2;                                                         //O(1)
}

int Unimodal(vector<int> &v, int ini, int fin){
  if (ini == fin){//Caso base 1. Sólo tenemos 1 elemento                 //O(1)
    return ini;                                                          //O(1)
  }
  else if (ini+1 == fin){//Caso base 2. Tenemos 2 elementos              //O(1)
    return posMax(v,ini,fin);                                            //O(1)
  }
  else{//Tenemos más de dos elementos
    int medio = (fin+ini)/2, cuarto,primera_pos,segunda_pos, maximo;     //O(1)

    if(v[medio]<v[medio-1]){//El pico está a la izquierda de medio       //O(1)
      cuarto = (medio+ini)/2;//Si tenemos 3 elementos, cuarto == ini     //O(1)
      primera_pos = Unimodal(v,ini,cuarto);                              //O(n/4)
      segunda_pos = Unimodal(v,cuarto+1,medio);                          //O(n/4)
    }
    else if(v[medio]<v[medio+1]){//El pico está a la derecha de medio    //O(1)
      cuarto = (fin+medio)/2;//Si tenemos 3 elementos, cuarto == medio   //O(1)
      primera_pos = Unimodal(v,medio,cuarto);                            //O(n/4)
      segunda_pos = Unimodal(v,cuarto+1,fin);                            //O(n/4)
    }
    else//Caso base 3.El pico es medio
      return medio;                                                      //O(1)

    maximo = posMax(v,primera_pos,segunda_pos);                          //O(1)

    return maximo;                                                       //O(1)
  }
}
```

Cálculo de Eficiencia Teórica

Para realizar el cálculo de la eficiencia teórica, hacemos uso de la fórmula maestra:

$$T(n) = I \cdot T\left(\frac{n}{b}\right) + G(n)$$

Mediante la cual y considerando $G(n)$ constante, y las siguientes condiciones:

- Número de subproblemas=4
- Dificultad de dividir y combinar=0

Obtenemos:

$$T(n) = 2 \cdot T\left(\frac{n}{4}\right) + 1$$
$$\downarrow$$
$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Teniendo en cuenta que si $I = b^k$, esto quiere decir que $O(n^k \log(n))$, y como $k = 0 \rightarrow n^k = 1$, podemos concluir que $O(\log(n))$, bastante más eficiente que el anterior.

Cálculo de Eficiencia Híbrida

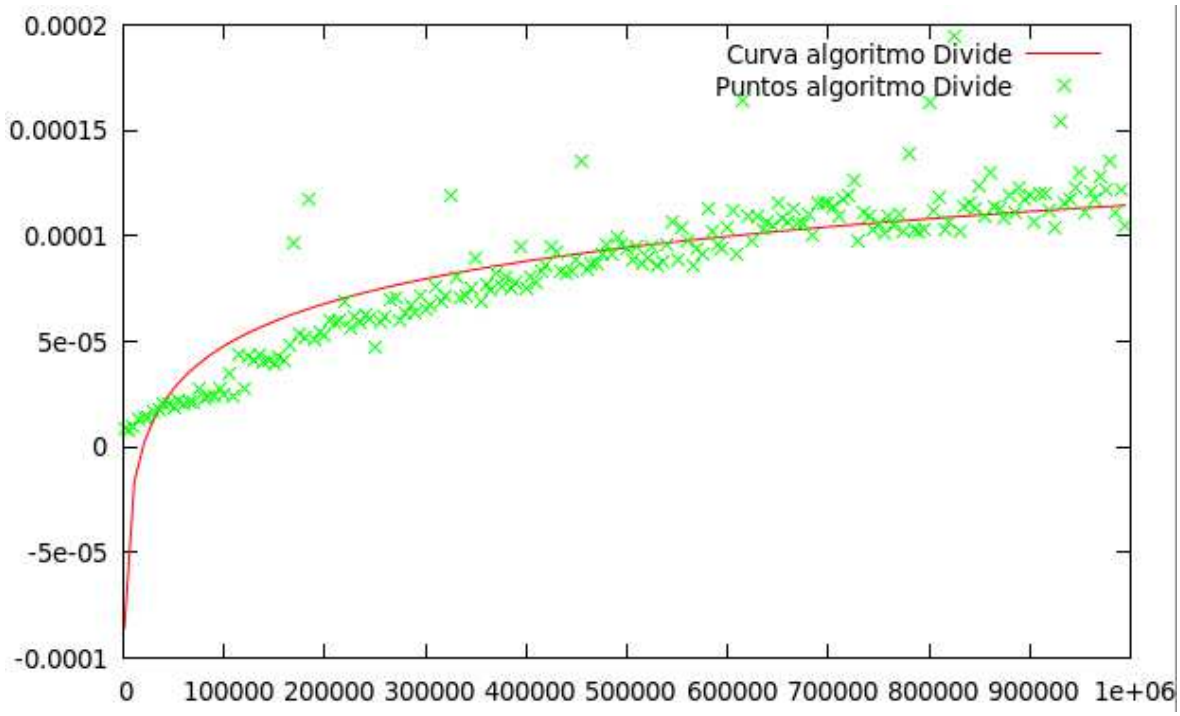
Procedemos a calcular su eficiencia empírica e híbrida. Para la empírica hemos usado el siguiente script:

```
1 #!/bin/csh -vx
2 @ i = 100
3 while ( $i < 20000000 )
4 ./algoritmoDivide $i >> divide.dat
5 @ i += 100000
6 end
```

Metemos los datos calculados en gnuplot.
Utilizamos la función: $g(x) = a_2 \cdot \log(x) + a_3$, y calculamos a partir de ellos sus constantes ocultas:

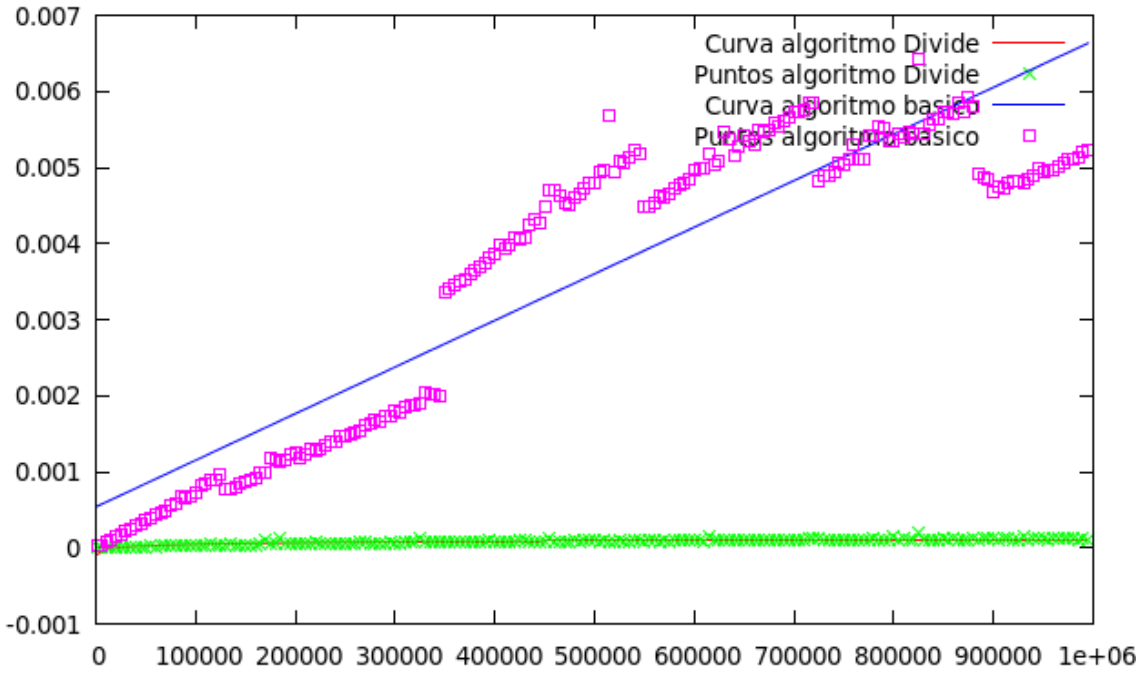
```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================

a2              = 2.90271e-05      +/- 1.154e-06    (3.977%)
a3              = -0.000286357     +/- 1.482e-05    (5.176%)
```

El resultado de la eficiencia híbrida y empírica se encuentra en la siguiente grafica:



Recordemos que es normal que los puntos estén dispersos, ya que eso depende de dónde esté situado el pico.

Por último, vamos a comprobar las eficiencias de ambos algoritmos, a ver si es verdad que se cumple que el algoritmo divide y vencerás es más eficiente que el obvio, valiéndonos de una gráfica con los dos algoritmos:



Queda demostrado que para este problema en particular es infinitamente más eficiente un algoritmo Divide y Vencerás que el haber creado un algoritmo normal, como bien puede apreciarse.