Práctica 3. Divide y vencerás

Moises Guerrero Lopez
moises.guerrerolopez@alum.uca.es
Teléfono: xxxxxxxx

5 de enero de 2019

1. Describa las estructuras de datos utilizados en cada caso para la representación del terreno de batalla.

   He creado una estructura llamada Casilla compuesta por por los atributos fila y col de tipo entero que representan la fila y la columna del campo de batalla y el atributo valor de tipo flotante que almacena la valoración obtenida mediante defaultCellValue según la fila y la columna del campo de batalla. Con esta estructura he creado un vector Casilla en la que almaceno la fila, columna y valoración de cada casilla del campo de batalla, de esta forma podemos ordenar muy fácilmente el vector, en función de su valoración, y podemos acceder muy fácilmente a el también.

2. Implemente su propia versión del algoritmo de ordenación por fusión. Muestre a continuación el código fuente relevante.

```
void fusion(Casilla* v,int low,int mid,int high,int size)
{

        int h,i,j,k;
        Casilla w[size];
        h=low;
        i=low;
        j=mid+1;

        //Esta parte es la que ordena el vector apoy ndose en un vector auxiliar

        //Primero ordena empezando por la mitad inferior
        while((h<=mid)&&(j<=high))
        {

                if(v[h].valor<=v[j].valor)
                {

                        w[i]=v[h];
                        ++h;

                }
                else
                {

                        w[i]=v[j];
                        ++j;
                }
                ++i;

        }

        //Y despues por la otra mitad
        if(h>mid)
        {
                for(k=j;k<=high;++k)
                {
                        w[i]=v[k];
                        ++i;
```

```
                }
        }
        else
        {
                for(k=h;k<=mid;k++)
                {
                        w[i]=v[k];
                        ++i;
                }
        }
        //Aqui devuelve los valores ya ordenados al vector original
        for(k=low;k<=high;++k)v[k]=w[k];
}


void sort_fusion(Casilla* defensas,int low,int high,int size)
{

        int mid;
        if(low<high)
        {

                mid=low+(high-low)/2;

                sort_fusion(defensas,low,mid,size);
                sort_fusion(defensas,mid+1,high,size);
                fusion(defensas,low,mid,high,size);

        }

}
```

3. Implemente su propia versión del algoritmo de ordenación rápida. Muestre a continuación el código fuente relevante.

```
int pivote(Casilla* posiciones,int low,int high)
{
        float p=posiciones[high].valor;
        int i=(low-1);
        Casilla aux;
        for(int j=low;j<=high-1;++j)
        {
                if(posiciones[j].valor<=p)
                {
                        ++i;
                        aux=posiciones[i];
                        posiciones[i]=posiciones[j];
                        posiciones[j]=aux;
                }
        }
        aux=posiciones[i+1];
        posiciones[i+1]=posiciones[high];
        posiciones[high]=aux;

        return i+1;
}

void quicksort(Casilla* posiciones,int low,int high)
{
        int p;
        if(low<high)
        {
                p=pivote(posiciones,low,high);
                quicksort(posiciones,low,p-1);
                quicksort(posiciones,p+1,high);
        }
}
```

4. Realice pruebas de caja negra para asegurar el correcto funcionamiento de los algoritmos de ordenación implementados en los ejercicios anteriores. Detalle a continuación el código relevante.

```
int main()
{
        //En este caso lo hacemos con enteros, asi que hemos tenido que modificar los metodos
             para que acepten enteros y no la estructurca Casilla

        int size=8;
        std::vector<int> A;
        A.push_back(1);A.push_back(12);A.push_back(5);A.push_back(17);A.push_back(8);A.
            push_back(90);A.push_back(99);A.push_back(5);
        std::vector<int> B;
        B.push_back(1);B.push_back(12);B.push_back(5);B.push_back(17);B.push_back(8);B.
            push_back(90);B.push_back(99);B.push_back(5);

        sort_fusion(A,0,size-1,size);
        quicksort(B,0,size-1);

        for(int i=0;i<size;++i)cout<<" "<<A[i];
        cout<<endl;
        for(int i=0;i<size;++i)cout<<" "<<B[i];
        cout<<endl;

        //Muestra
        //1 5 5 8 12 17 90 99
        //1 5 5 8 12 17 90 99
}
```

5. Analice de forma teórica la complejidad de las diferentes versiones del algoritmo de colocación de defensas en función de la estructura de representación del terreno de batalla elegida. Comente a continuación los resultados. Suponga un terreno de batalla cuadrado en todos los casos.

   Para realizar el análisis tenemos en cuenta que l es el número de defensas y n es el tamaño del mapa, anchura*altura, ya que hemos usado un vector. Para la colocación sin ordenar previamente las defensas obtenemos una complejidad de t(l)=n+l*(n*2*l)
   Como tenemos que buscar la mejor posición donde colocar cada defensa, tenemos que recorrer el vector l veces y la función de factibilidad por cada defensa
   Para la ordenación por fusión la cosa mejora algo, tiene una complejidad de

$$t(l) = n + (g * \log g) + \sum_{i=0}^{i=l} l * (i * 2 * l)$$

   Al haber pre ordenado previamente, con una complejidad de n*log n, hace que los pasos restantes sean solo contar el numero de veces que hay que realizar la función de factibilidad. Donde

$$g = n0 * 2^k$$

   como se indico en la clase teórica.
   Para la ordenación rápida, como su peor caso es de orden

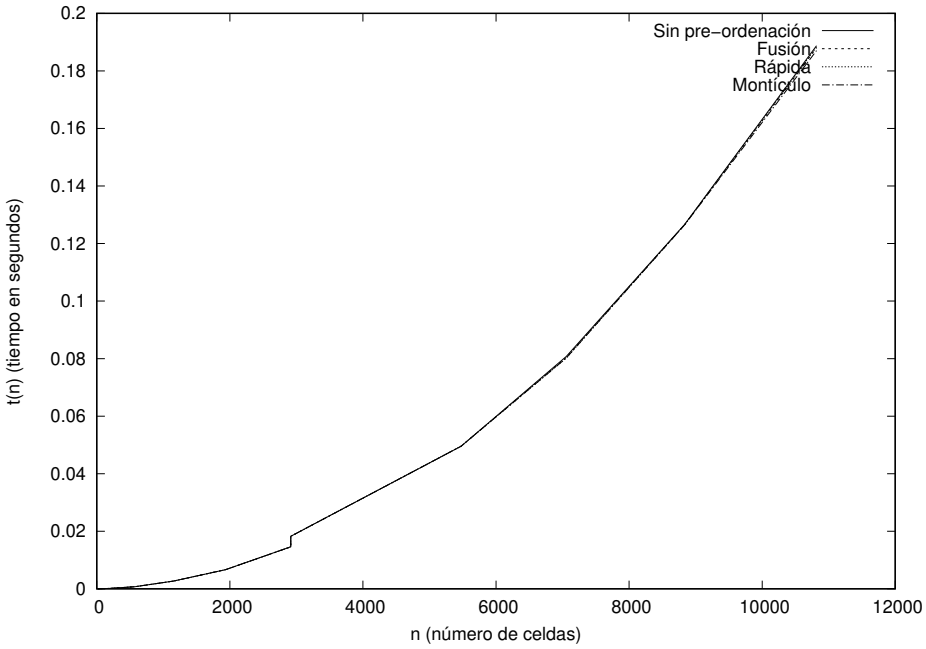
$$n^2$$

   notamos un empeoramiento bastante considerable en comparación con la ordenación por fusión y muy similar a la ordenación sin pre ordenar.
   Tiene una complejidad de

$$t(l) = n + \frac{n(n-1)}{2} + \sum_{i=0}^{i=l} l * (i * 2 * l)$$

   .
   Y finalmente con la ordenación mediante montículo, que en nuestro caso hemos usado una estructura de datos para la representación de un montículo, vemos una complejidad similar al de ordenación por fusión,pero ligeramente mejor.
   Tiene una complejidad de

$$t(l) = n + n + n * log * n + \sum_{i=0}^{i=l} l * (i * 2 * l)$$



6. Incluya a continuación una gráfica con los resultados obtenidos. Utilice un esquema indirecto de medida (considere un error absoluto de valor 0.01 y un error relativo de valor 0.001). Es recomendable que diseñe y utilice su propio código para la medición de tiempos en lugar de usar la opción *-time-placeDefenses3* del simulador. Considere en su análisis los planetas con códigos 1500, 2500, 3500,..., 10500, al menos. Puede incluir en su análisis otros planetas que considere oportunos para justificar los resultados. Muestre a continuación el código relevante utilizado para la toma de tiempos y la realización de la gráfica.

Todo el material incluido en esta memoria y en los ficheros asociados es de mi autoría o ha sido facilitado por los profesores de la asignatura. Haciendo entrega de este documento confirmo que he leído la normativa de la asignatura, incluido el punto que respecta al uso de material no original.