Práctica 3. Divide y vencerás

Alejandro Serrano Fernandez
ale.serranofer@alum.uca.es
Teléfono: 640217690
NIF: 20501318S

20 de diciembre de 2020

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

   Para representar el terreno de batalla he hecho uso de un un struct, llamado Celda donde almacenaremos las posiciones de la fila y columna, junto con el valor determinado por la función defaultCellValue.

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

```
void ordenacion_fusion(std::vector<Celda>& v, int i, int j)
{
        int n = j - i + 1;
        int k = 0;

        if(n <= 3)
                ordenacion_insercion(v,i,j);
        else
        {
                k = i - 1 + n/2;
                ordenacion_fusion(v,i,k);
                ordenacion_fusion(v, k+1, j);
                fusion(v,i,k,j);
        }
}



void fusion(std::vector<Celda>& v, int i, int k, int j)
{
        int n = j - i + 1;
        int p = i;
        int q = k;
        std::vector<Celda> w;

        for(int l = 0; l < n; l++)
        {
                if(p < k && (q > j-1 || v[p] <= v[q]))
                {
                        w.push_back(v[p]);
                        p = p + 1;
                }
                else
                {
                        w.push_back(v[q]);
                        q = q + 1;
                }
        }

        for(int l = 0; l < n; l++)
        {
                v[i + l] = w[l];
        }
}
```

```
void ordenacion_insercion(std::vector<Celda>& v, int i, int j)
{
        int k;
        Celda aux;

        for(int t = i; t <= j; t++)
        {

                aux = v[t];
                for(k = t; k > 0 && (aux < v[k-1]); k--)
                {
                        v[k] = v[k-1];
                }
                v[k] = aux;
        }
}
```

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

```
void ordenacion_rapida(std::vector<Celda>& v, int i, int j)
{
        int p;
        int n = j - i + 1;

        if(n <= 3)
        {
                ordenacion_insercion(v,i,j);
        }
        else
        {
                p = pivote(v,i,j);
                ordenacion_rapida(v,i,p-1);
                ordenacion_rapida(v,p+1,j);
        }
}

int pivote(std::vector<Celda>& v, int i, int j)
{
        int p = i;
        Celda x = v[i];
        Celda aux;
        for(int k=i+1;k < j;k++)
        {
                if(v[k].value_ <= x.value_)
                {
                        p=p+1;
                        aux = v[p];
                        v[p] = v[k];
                        v[k] = aux;
                }
        }

        v[i] = v[p];
        v[p] = x;

        return p;
}

void ordenacion_insercion(std::vector<Celda>& v, int i, int j)
{
        int k;
        Celda aux;

        for(int t = i; t <= j; t++)
        {

                aux = v[t];
                for(k = t; k > 0 && (aux < v[k-1]); k--)
```

```
                {
                        v[k] = v[k-1];
                }
                v[k] = aux;
        }
}
```

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.

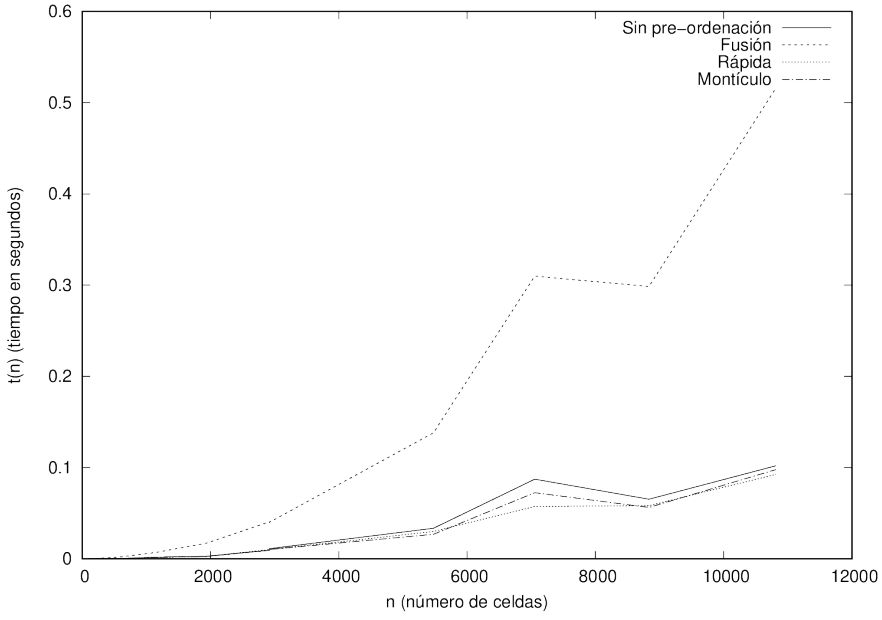
```
void caja_negra(const std::vector<Celda>& v)
{
        Celda aux;
        bool fail = false;
        std::vector<Celda> v2 = v;


        /*      SIN ORDENACION          */

        std::next_permutation(v2.begin(),v2.end()); //Pasa a la siguiente permutacion

        while(!v2.empty())
        {
                sin_ordenacion(v2);

                aux = v2.back();
                v2.pop_back();

                //Comprobamos que el elemento anterior sacado es mayor que el ultimo elemento
                if(!v2.empty())
                {
                        sin_ordenacion(v2);
                        if( aux.value_ < v2.back().value_ )
                        {
                                fail = true;
                        }

                }
        }

        if(!fail)
                std::cout<<"Sin ordenacion realizado correctamente"<<std::endl;
        else std::cout<<"ERROR: Sin ordenacion ha fracasado"<<std::endl;


        v2 = v;
        std::next_permutation(v2.begin(),v2.end());


        /*      ORDENACION POR FUSION   */
        ordenacion_fusion(v2, 0, v.size()-1);

        if(comprobar_ordenado(v2))
                std::cout<<"Con fusion esta ordenado"<<std::endl;
        else std::cout<<"ERROR: Con fusion no esta ordenado"<<std::endl;



        v2 = v;
        std::next_permutation(v2.begin(),v2.end());


        /*      ORDENACION RAPIDA       */
        ordenacion_rapida(v2, 0, v.size()-1);

        if(comprobar_ordenado(v2))
                std::cout<<"Con ord.rapida esta ordenado"<<std::endl;
        else std::cout<<"ERROR: Con ord.rapida no esta ordenado"<<std::endl;
```

```
        v2 = v;
        std::next_permutation(v2.begin(),v2.end());


        /*      ORDENACION POR MONTICULO        */
        ordenacion_monticulo(v2);

        if(comprobar_ordenado(v2))
                std::cout<<"Con monticulo esta ordenado"<<std::endl;
        else std::cout<<"ERROR: Con monticulo no esta ordenado"<<std::endl;

}


//Comprobamos que el vector esta ordenado de menor a mayor
bool comprobar_ordenado(const std::vector<Celda>& v)
{
        bool ordenado = true;

        for(int i = 1; i != v.size() - 1; i++)
        {
                if(v[i-1].value_ > v[i].value_)
                        ordenado = false;
        }

        return ordenado;
}
```

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.

   Escriba aquí su respuesta al ejercicio 5.

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.