Práctica 3. Divide y vencerás

Manuel Diaz Gil
manuel.diazgil@alum.uca.es
Teléfono: 667361517
NIF: 45382945N

8 de enero de 2018

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

   He usado una estructura celda en la que guardo la posicion de la fila y la columna, junto al valor que le da el cellValue. Tiene definido todos los constructores necesarios y operadores de comparacion, y usa el de asignacion por defecto.

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

```
void ordenacionInsercion(std::vector<celda>& c, int i, int k){
    int j;
    celda aux;
    for(i = 0;i<k;i++){
        aux = c[i];
        for(j = i;j>0 && (aux<c[j-1]);j--){
                c[j]=c[j-1];
        }
        c[j]=aux;
    }
}

void fusion(std::vector<celda>& c,int i,int k,int j){
    int n = j-i;
    std::vector<celda> aux;
    int iter=i;
    int iter2=k;
    for (int l = 0; l < n; ++l)
    {
        if(iter<=k && (iter2>j-1 || c[iter]<=c[iter2])){
            aux.push_back(c[iter]);
            ++iter;
        }
        else{
            aux.push_back(c[iter2]);
            ++iter2;
        }
    }
    for (int l = 0; l < n; ++l)
    {
        c[l+i]=aux[l];
    }
}

void ordenacionFusion(std::vector<celda>& c,int i, int j){
    int n = j-i;
    if (n<3){
        ordenacionInsercion(c,i,j);
    }
    else{
        int k=j-((j-i)/2);
        ordenacionFusion(c,i,k);
        ordenacionFusion(c,k,j);
        fusion(c,i,k,j);
```

```
    }

}
```

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

```
void ordenacionInsercion(std::vector<celda>& c, int i, int k){
    int j;
    celda aux;
    for(i = 0;i<k;i++){
        aux = c[i];
        for(j = i;j>0 && (aux<c[j-1]);j--){
                c[j]=c[j-1];
        }
        c[j]=aux;
    }
}

int pivote(std::vector<celda>& c,int i, int j){
    int p = i;
    celda x = c[i];
    for (int k = i+1; k < j; ++k)
    {
        if(c[k]<=x){
            ++p;
            std::swap(c[p],c[k]);
        }
    }
    c[i] = c[p];
    c[p] = x;
    return p;
}

void ordenacionRapida(std::vector<celda>& c,int i, int j){
    int n = j-i;
    if(n<3){
        ordenacionInsercion(c,i,j);
    }else{
        int p = pivote(c,i,j);
        ordenacionRapida(c,i,p-1);
        ordenacionRapida(c,p+1,j);
    }
}
```

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.

```
bool compruebaOrdenado(std::vector<celda>& c){
    for (int i = 0; i < c.size()-1; ++i)
    {
        if(c[i]>c[i+1]){
            return false;
        }
    }
    return true;
}

void pruebaCajanegra(std::vector<celda>& c){
    std::vector<celda> aux(c);
    if (compruebaOrdenado(aux))
    {
        std::cout<<" El vector esta ordenado"<<std::endl;
    }else{
        std::cout<<" El vector no esta ordenado"<<std::endl;
    }

    sinOrdenacion(aux);
```

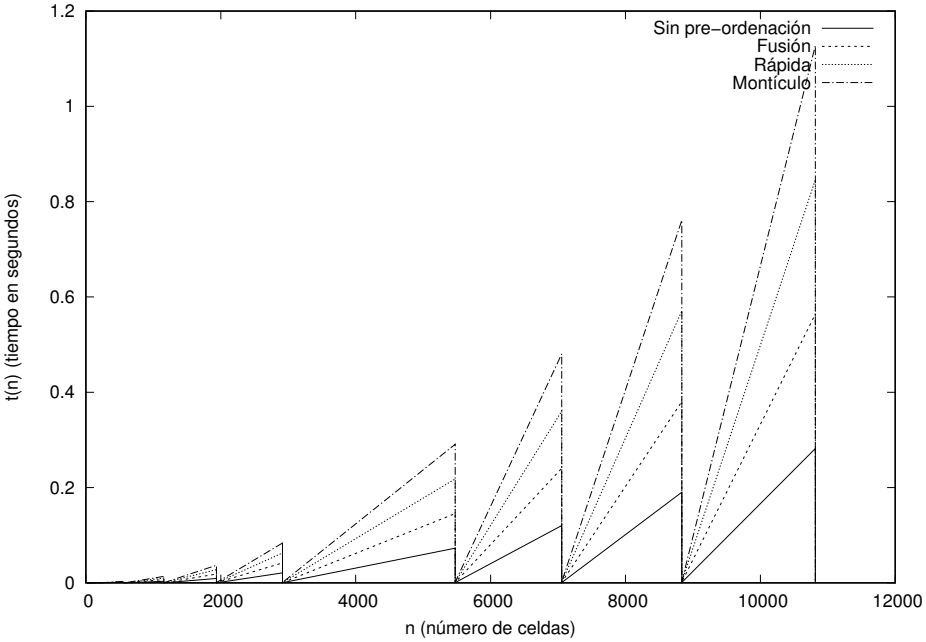
```
    if (compruebaOrdenado(aux))
    {
        std::cout<<"Sin ordenacion: El vector esta ordenado"<<std::endl;
    }else{
        std::cout<<"Sin ordenacion: El vector no esta ordenado"<<std::endl;
    }
    aux=c;
    ordenacionFusion(aux,0,aux.size());
    if (compruebaOrdenado(aux))
    {
        std::cout<<"ordenacionFusion El vector esta ordenado"<<std::endl;
    }else{
        std::cout<<"ordenacionFusion El vector no esta ordenado"<<std::endl;
    }
    aux=c;
    ordenacionRapida(aux,0,aux.size());
    if (compruebaOrdenado(aux))
    {
        std::cout<<"ordenacionRapida: El vector esta ordenado"<<std::endl;
    }else{
        std::cout<<"ordenacionRapida: El vector no esta ordenado"<<std::endl;
    }
    aux=c;
    ordenacionMonticulo(aux,0,aux.size());
    if (compruebaOrdenado(aux))
    {
        std::cout<<"ordenacionMonticulo: El vector esta ordenado"<<std::endl;
    }else{
        std::cout<<"ordenacionMonticulo: El vector no esta ordenado"<<std::endl;
    }


}
```

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.

   Siendo n el tamaño de la matriz y p el numero de casillas que se valoran.Sin ordenacion la complejidad es de Orden n elevado p y los demas algoritmos son de Orden nlog(n).

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). Considere en su análisis los planetas con códigos 1500, 2500, 3500,..., 10500. Incluya en el análisis los planetas que considere oportunos para mostrar información relevante.

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.


Sin pre−ordenación
Fusión
Rápida
Montículo
t(n) (tiempo en segundos)
n (número de celdas)
0
0.2
0.4
0.6
0.8
1
1.2
0
2000
4000
6000
8000
10000
12000