Práctica 3. Divide y vencerás

Alejandro Segovia Gallardo
alejandro.segoviagallardo@alum.uca.es
Teléfono: 608842858
NIF: 32083695Y

8 de enero de 2018

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

   He utilizado un struct Defensas que he creado para el almacenamiento de la puntuacion de las celdas y su disposicion en el tablero, por otro lado he usado un vector de la estructura anteriormente mencionada, a la cual le he aplicados los algoritmos pertienentes cuando ha sido necesario.

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

```
std::vector<Defensas> fusion(const std::vector<Defensas>& v1,
                          const std::vector<Defensas>& v2)
{
    std::vector<Defensas> aux(v1.size() + v2.size());
    int i, j, k;
    i = j = k = 0;
    while(i < v1.size() and j < v2.size()) {
        if(v1[i].puntuacion <= v2[j].puntuacion) {
            aux[k].puntuacion = v1[i].puntuacion;
            ++i;
        }
        else {
            aux[k].puntuacion = v2[j].puntuacion;
            ++j;
        }
        ++k;
    }
    while(i < v1.size()) {
        aux[k].puntuacion = v1[i].puntuacion;
        ++k;
        ++i;
    }
    while(j < v2.size()) {
        aux[k].puntuacion = v2[j].puntuacion;
        ++k;
        ++j;
    }
    return aux;
}

void ordenacionFusion(std::vector<Defensas>& v)
{
  int m = v.size()/2;
  if(m != 0) {
      std::vector<Defensas> aux1(m);
      for(int i = 0; i < aux1.size() - 1; ++i)
          aux1[i].puntuacion = v[i].puntuacion;

      std::vector<Defensas> aux2(v.size()-aux1.size());
      for(int i = 0; i < aux2.size() - 1; ++i)
        aux1[i].puntuacion = v[aux1.size()+i].puntuacion;

        ordenacionFusion(aux1);
        ordenacionFusion(aux2);
```

```
        v = fusion(aux1, aux2);
  }
}
```

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

```
int divide(std::vector<Defensas>& array, int start, int end) {
  int p = start;
  float x = array[start].puntuacion;
  for(int i = start+1; i<=end; i++)
    if(array[i].puntuacion <= x){
      p++;
      Defensas aux = array[i];
      array[i] = array[p];
      array[p] = aux;
    }
  Defensas array_aux = array[start];
  array[start] = array[p];
  array[p].x = array_aux.x;
  array[p].y = array_aux.y;
  array[p].puntuacion = x;
}


void quicksort(std::vector<Defensas>& array, int start, int end)
{
    int n = end - start + 1;
    if(n <= 2){
      std::sort(array.begin(),array.end());
    }
    else{

        int pivot = divide(array, start, end);

        // Ordeno la lista de los menores
        quicksort(array, start, pivot - 1);

        // Ordeno la lista de los mayores
        quicksort(array, pivot + 1, end);
    }
}
```

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.

   Pruebas cajas negras para Fusion:

```
//Codigo previo
ordenacionFusion(v);
//Codigo previo
for(int i = 0;i != v.size() - 1; i++ ){
  if(v[i].puntuacion > v[i+1].puntuacion)
    cajaNegraFusion = true; //Estar  a true cuando falle el algoritmo
```

   Pruebas caja negra Ordenacion Rapida:

```
//Codigo previo
quicksort(v,0,v.size()-1);
//Codigo previo
for(int i = 0;i != v.size() - 1; i++){
  if(v[i].puntuacion > v[i+1].puntuacion)
    cajaNegraRapida = true; //Estar  a true cuando falle el algoritmo
}
```

   Pruebas caja negra Monticulo:

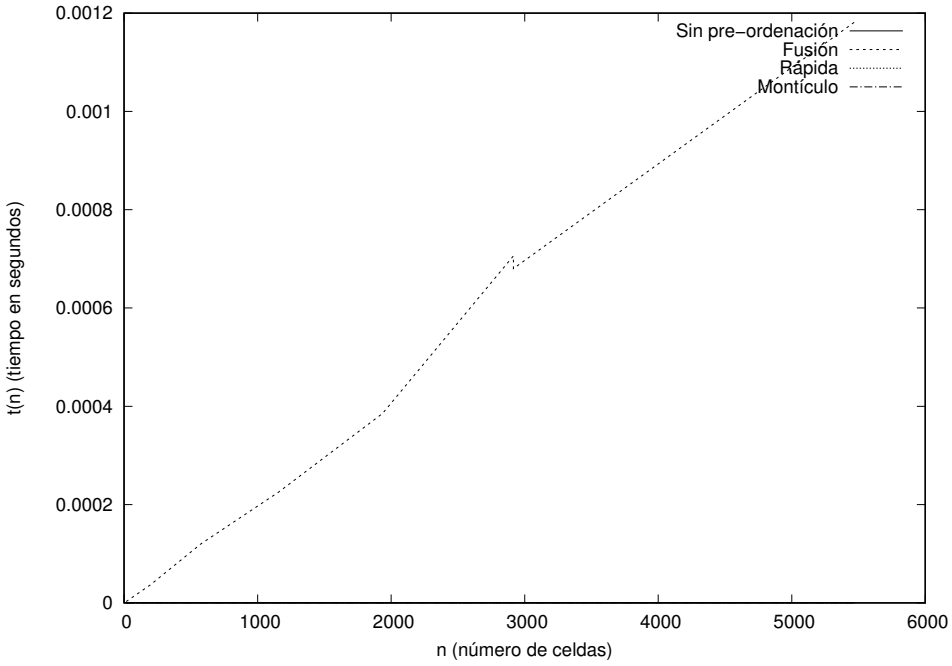
```
std::make_heap(v.begin(),v.end());
std::sort_heap(v.begin(),v.end());
for (size_t i = 0; i < v.size()-1; i++) {
  if(v[i].puntuacion > v[i+1].puntuacion)
    cajaNegraMonticulo= true; //Estar a true cuando falle el algoritmo
}
```

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 el caso de sin preordenacion, el codigo para colocar las defensas es de orden $\theta(n^2)$. Para elmontículo, el algoritmo de colocación de defensas es de orden $\theta(n)$. Aunque para los de fusion y ordenacion rápida, será de n*$\cdot log_2(n)$.

   NOTA: Para los 3 casos estoy obviando la medida adaptativa(bucle do while)

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.