# Práctica 3. Divide y vencerás

Carmen del Mar Ruiz de Celis
carmen.ruizdecelis@alum.uca.es
Teléfono: xxxxxxxx
NIF:49565250C

6 de enero 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 -en todos los casos-, he optado por utilizar una matriz o vector de vectores; ya que, era para mí, mucho más cómodo a la hora de comprobar posiciones válidas y darle valores. En cualquier caso, poco importa la estructura "original", ya que con lo que realmente se trabaja luego es con una copia, para lo que he elegido un vector de tipo "cellValueStruct", la cual almacena el valor y las coordenadas x e y, aparte de definir sus propios operadores.

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

```
void fusion(std::vector<cellValueStruct> &v, int i, int j, int k){
    int left= k-i+1, right= j-k, iaux = 0, jaux = 0, kaux = i;

    std::vector<cellValueStruct> L(left), R(right);

    for (int r = 0; r < right; ++r) R[r] = v[k+1+r];
    for (int x = 0; x < left; ++x) L[x] = v[x+i];

    while (iaux < left && jaux < right){
        if(L[iaux] <= R[jaux]){
            v[kaux] = L[iaux];
            iaux++;
        }
        else{
            v[kaux] = R[jaux];
            jaux++;
        }

        kaux++;
    }

    while (jaux < right){
        v[kaux] = R[jaux];
        jaux++;
        kaux++;
    }
    while (iaux < left){
        v[kaux] = L[iaux];
        iaux++;
        kaux++;
    }

}


void sortFusion(std::vector<cellValueStruct> &v, int i, int j){
    if (i < j) {
        int k = i+(j-i)/2;
        sortFusion(v, i, k);
        sortFusion(v, k+1, j);
        fusion(v, i, j, k);
```

```
    }
}
```

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

```
    int pivote(std::vector<cellValueStruct> &v, int start, int end)  {
    cellValueStruct pivot = v[end];
    int min = (start - 1);

    for (int i = start; i < end; i++){
        if (v[i] <= pivot){
            min++;
            std::swap(v[min], v[i]);
        }
    }
    std::swap(v[min + 1], v[end]);
    return (min + 1);
}


void quickSort(std::vector<cellValueStruct> &v, int i, int j){

    if (i < j) {
        int pivot = pivote(v, i, j);
        quickSort(v, i, pivot-1);
        quickSort(v, pivot+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.

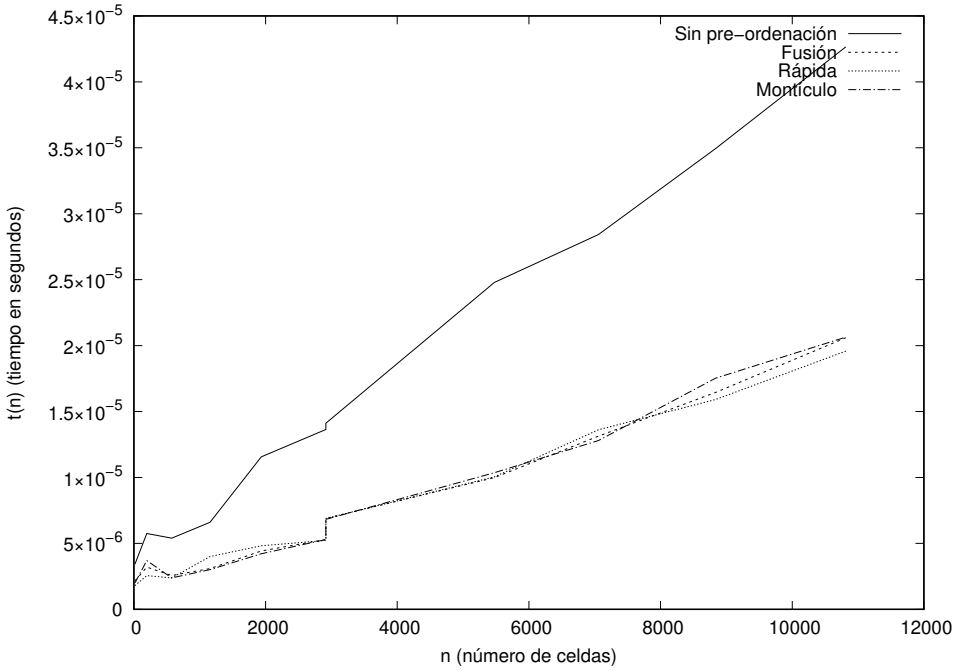
```
void generar(std::vector<int> &v){
    for(int i = 0; i < 15; i++) v.push_back(rand());
}

void comprobar(std::vector<int> &v){
     std::vector<int> ordenado = v;
     std::sort(ordenado.begin(), ordenado.end());
     if(std::equal(ordenado.begin(), ordenado.end(), v.begin()))
        std::cout << "Ha ordenado correctamente" << std::endl;
     else std::cout << "Ha ordenado mal" << std::endl;
}

int main(){

    std::vector<int> v;
    std::cout << "QUICKSORT" << std::endl;
    generar(v);
    quickSort(v, 0, v.size()-1);
    comprobar(v);

    std::cout << "FUSION" << std::endl;
    generar(v);
    sortFusion(v, 0, v.size()-1);
    comprobar(v);

}
```

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.

   - En el algoritmo sin ordenación, la función de selección busca para cada defensa la mejor celda a la que se puede optar, y en el peor caso ésta sería la última. Luego, la complejidad sería, suponiendo que n es el número de defensas y m el número de celdas del mapa, $(n^m)$.

- Para la ordenación por fusión, la estructura se divide por la mitad, quedando dos partes de aproximadamente n/2. Ambas se ordenan de manera recursiva y luego han de combinarse, lo que requiere de n comparaciones. Luego, tendría una complejidad de t(n/2) + t(n/2) + n. En el caso de llegar a su tamaño mínimo, se realizaría una inserción directa, iterando n(n-1)/2 veces. Luego es de complejidad ($n^2$).
- En el de ordenación rápida, toma un elemento como pivote y reorganiza elsubvector para que a la izquierda del pivote queden los menores o iguales, y a su derecha los mayores. Luego, el peor caso se produce cuando el pivote está en un extremo, realizando así n(n-1)/2 iteraciones, siendo de complejidad ($n^2$).
- Según la propiedad de completitud y orden del montículo, el orden de ordenar es el equivalente al de inserción, que es de n log n (de acuerdo con la implementación del árbol que utiliza la librería de C++), ya que la altura del árbol es de log n y siempre inserta en el último nivel.

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.




```
placeDefenses3(bool** freeCells, int nCellsWidth, int nCellsHeight, float mapWidth, float
     mapHeight
, List<Object*> obstacles, List<Defense*> defenses) {

    float cellWidth = mapWidth / nCellsWidth;
    float cellHeight = mapHeight / nCellsHeight;
    cronometro cNoOrder, cQuickSort, cFusion, cHeap;
    int rNoOrder = 0, rFusion = 0, rQuickSort = 0, rMonticulo = 0;

    for(int it = 0; it<4; it++){

        std::vector<std::vector<float>> MapValue(nCellsHeight, std::vector<float>(
            nCellsWidth));
        int maxAttemps = 1000;
        std::vector<cellValueStruct> cells;

        for(int i = 0; i < nCellsHeight; i++){
            for(int j = 0; j < nCellsWidth; j++){
```

```
MapValue[i][j] = defaultCellValue(i, j, nCellsWidth, nCellsHeight,
    mapWidth, mapHeight, obstacles, defenses);
}
}


for(int i = 0; i < nCellsHeight; i++){
    for(int j = 0; j < nCellsWidth; j++){
        cells.push_back(cellValueStruct(i, j, MapValue[i][j]));
        }
        }


        switch(it){
            case 0:
            cNoOrder.activar();
            sort(cells.begin(), cells.end(), compare());

            case 1:
            cFusion.activar();
            sortFusion(cells, 0, cells.size()-1);

            break;
            case 2:
            cQuickSort.activar();
            quickSort(cells, 0, cells.size()-1);

            break;
            case 3:
            cHeap.activar();
            heap(cells);

            break;
            }


            Vector3 v;
            int i = cells.size()-1;
            int row, col;
            bool condition=true;

            for( List<Defense*>::iterator currentDefense = defenses.begin
                (); currentDefense != defenses.end() && maxAttemps > 0 &&
                 condition; maxAttemps--, i--){

                if(it!=0) v = cellCenterToPosition(cells[i].x, cells[i].y
                    , cellWidth, cellHeight);
                else v = selection(cells, cellWidth, cellHeight);
                positionToCell(v, row, col, cellWidth, cellHeight);
                if(feasibility((*currentDefense)->id, row, col,
                    nCellsWidth, nCellsHeight, mapWidth, mapHeight,
                    obstacles, defenses)){
                    maxAttemps = 1000;
                    (*currentDefense)->position = v;
                    cells.erase(cells.begin()+i);
                    i=cells.size()-1;
                    currentDefense++;
                    }


                    switch(it){
                        case 0 :
                        rNoOrder++;
                        condition = cNoOrder.tiempo() < (0.01 / (0.001 +
                            0.01));
                        break;
                        case 1:
                        rFusion++;
                        condition = cFusion.tiempo() < (0.01 / (0.001 +
                            0.01));
                        break;
```

```
case 2:
rQuickSort++;
condition = cQuickSort.tiempo() < (0.01 / (0.001
    + 0.01));
break;
case 3:
rMonticulo++;
condition = cHeap.tiempo() < (0.01 / (0.001 +
    0.01));
break;
}

}



switch(it){
    case 0:
    cNoOrder.parar();
    case 1:
    cFusion.parar();
    break;
    case 2:
    cQuickSort.parar();
    break;
    case 3:
    cHeap.parar();
    break;
    }

    if(it!=0) cells.clear();
    }

    std::cout << (nCellsWidth * nCellsHeight)
    << '\t' << cNoOrder.tiempo() / rNoOrder
    << '\t' << cFusion.tiempo() / rFusion
    << '\t' << cQuickSort.tiempo() / rQuickSort
    << '\t' << cHeap.tiempo() / rMonticulo
    << std::endl;
    }
```

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.