Práctica 3. Divide y vencerás

Iván Alba Gómez
ivan.albagomez@alum.uca.es
Teléfono: XXXXXXXXX
NIF: 49302616T

8 de diciembre de 2021

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

   Para la representación del terreno de batalla he utilizado un vector de Cell, cuya estructura es la siguiente:

```
class Cell {
    public:
        Cell(int row = 0, int col = 0, float value = 0) : row_(row), col_(col), value_(value)
             {}
        int getRow() { return row_; }
        int getCol() { return col_; }
        float getValue() { return value_; }
    private:
        int row_, col_;
        float value_;
};
```

   Por lo tanto:

```
std::vector<Cell> cellVector;
```

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

```
void insertionSort(std::vector<Cell>& cellVector, int i, int j) {
    Cell cell;
    for (int l = i + 1; l <= j; l++) {
        cell = cellVector[l];
        int p = l - 1;
        while (p >= i && cellVector[p].getValue() < cell.getValue()) {
            cellVector[p+1] = cellVector[p];
            p--;
        }
        cellVector[p+1] = cell;
    }
}

void fusion(std::vector<Cell>& cellVector, int i, int k, int j) {
    int n = j - i + 1;
    int p = i;
    int q = k + 1;
    std::vector<Cell> w(n);
    for(int l = 0; l < n; l++) {
        if(p <= k && (q > j || cellVector[p].getValue() <= cellVector[q].getValue())) {
            w[l] = cellVector[p];
            p++;
        } else {
            w[l] = cellVector[q];
            q++;
```

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

void fusionSort(std::vector<Cell>& cellVector, int i, int j) {
    int n = j - i + 1;
    int umbral = 2;
    if(n <= umbral) {
        insertionSort(cellVector, i, j);
    } else {
        int k = i -1 + (n / 2);
        fusionSort(cellVector, i, k);
        fusionSort(cellVector, k + 1, j);
        fusion(cellVector, i, k, j);
    }
}
```

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

```
void insertionSort(std::vector<Cell>& cellVector, int i, int j) {
    Cell cell;
    for (int l = i + 1; l <= j; l++) {
        cell = cellVector[l];
        int p = l - 1;
        while (p >= i && cellVector[p].getValue() < cell.getValue()) {
            cellVector[p+1] = cellVector[p];
            p--;
        }
        cellVector[p+1] = cell;
    }
}

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

void fastSort(std::vector<Cell>& cellVector, int i, int j) {
    int n = j - i + 1;
    int umbral = 2;
    if(n <= umbral) {
        insertionSort(cellVector, i, j);
    } else {
        int p = pivote(cellVector, i, j);
        fastSort(cellVector, i, p - 1);
        fastSort(cellVector, 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.

   Para las pruebas de caja negra se ha usado un vector de enteros, en vez de un vector de Cell, con el objetivo de simplificar las pruebas.

```
int main() {

    int N = 5;

    std::vector<int> v1;
    std::vector<int> v2;
    std::vector<int> v3;

    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            v1.push_back(rand() % 100);      // rand de 0 a 99
            v2.push_back(rand() % 100);      // rand de 0 a 99
            v3.push_back(rand() % 100);      // rand de 0 a 99
        }
    }

    std::cout << "VECTOR V1" << std::endl;
    imprimir(v1);
    fusionSort(v1, 0, v1.size() - 1);
    imprimir(v1);

    std::cout << "VECTOR V2" << std::endl;
    imprimir(v2);
    fastSort(v2, 0, v2.size() - 1);
    imprimir(v2);

    std::cout << "VECTOR V3" << std::endl;
    imprimir(v3);
    heapSort(v3, v3.size());
    imprimir(v3);
}
```

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.

   Complejidad de las diferentes versiones del algoritmo de colocación de defensas:

   - Sin preordenación: $O(n^2)$
   - Con preordenación por fusión: $O(nlog(n))$
   - Con preordenación rápida: $O(nlog(n))$
   - Con preordenación por montículos: $O(nlog(n))$

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). 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.

   Código para la toma de tiempos:

```
float E_ABS = 0.01;
float E_REL = 0.001;
cronometro c;
    long int r = 0;
    c.activar();
    do {
                List<Defense*>::iterator currentDefense = defenses.begin();
                while(currentDefense != defenses.end()) {
                        // C DIGO RELEVANTE
            currentDefense++;
                }
                r++;
```

```
    } while(c.tiempo() < E_ABS/E_REL+E_ABS);
    c.parar();
```

Para la realización de la gráfica, primero ejecutamos la orden *make data* para obtener los datos necesarios para la realización de la misma. Luego ejecutamos la orden *make plot* para que se genere la gráfica. (En mi caso no he hecho uso de esta instrucción, puesto que la he generado manualmente con los datos incluidos en 3 archivos diferentes. La orden que he usado ha sido *plot "tiemposFusion.txt" w l lw 3, "tiemposRapida.txt" w l lw 3, "tiemposMonticulo.txt" w l lw 3*)

```
tiemposFusion.txt
16      1.05307e-05
196     0.00395125
576     0.0447747
1156    0.305594
1936    1.22214
2916    3.68824
2916    5.24683
5476    22.8259
7056    46.9533
8836    89.7806
10816   161.777

tiemposRapida.txt
16      1.09583e-05
196     0.00380638
576     0.0467553
1156    0.324432
1936    1.24391
2916    4.18335
2916    5.27534
5476    22.6422
7056    47.0254
8836    90.4031
10816   161.983

tiemposMonticulo.txt
16      8.02297e-06
196     0.0036961
576     0.0455317
1156    0.317036
1936    1.28819
2916    4.32741
2916    5.47745
5476    23.4846
7056    46.4571
8836    91.6098
10816   164.344
```

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.

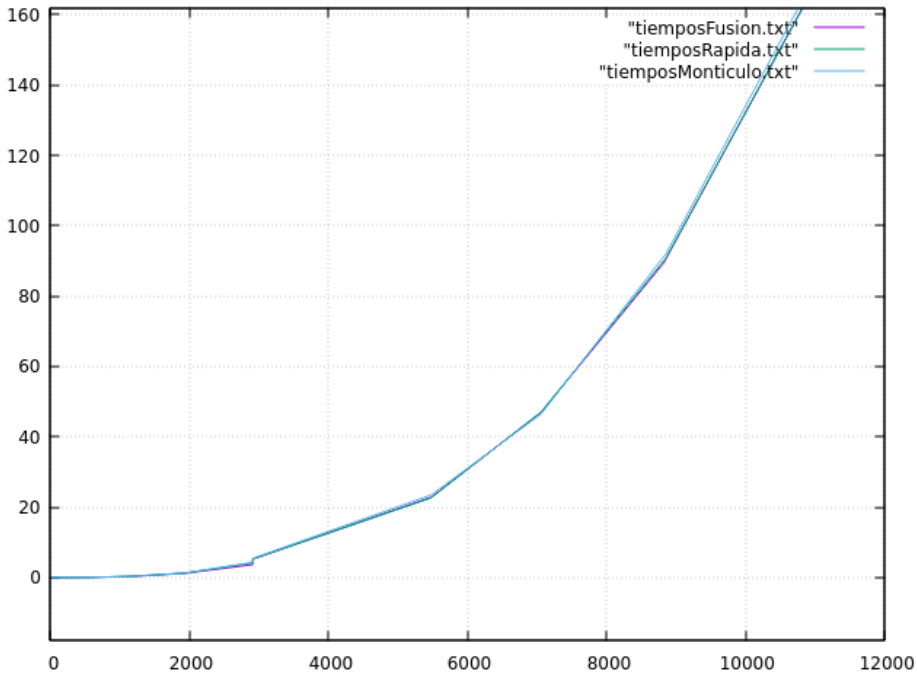

Figura 1: Tiempos