Práctica 3. Divide y vencerás

Lucía Atienza Olmo
lucia.atienzaolmo@alum.uca.es
Teléfono: 646767043
NIF: 32089267N

18 de diciembre de 2021

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

   En todos los casos se ha usado un struct celda para representar las casillas del mapa:

```
struct celda
{
    double valor;
    int row, col;
    celda() : valor(0), row(0), col(0) {}
    celda(int row, int col, double valor) : row(row), col(col), valor(valor) {}
    bool operator<(const celda& c) const
    {
        return valor < c.valor;
    }
    bool operator<=(const celda& c) const
    {
        return valor < c.valor || valor == c.valor;
    }
};
```

   Y para la representación del mapa se ha usado en todos los casos el contenedor de la biblioteca estándar std::vector.

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

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

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

void ordenacion_fusion(std::vector<celda>& v, int i, int j)
{
```

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

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

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

void ordenacion_rapida(std::vector<celda>& v, int i, int j)
{
    int n = j - i;
    if(n <= 3)
        insertionSort(v, i, j);
    else
    {
        int p = pivote(v, i, j);
        ordenacion_rapida(v, i, p);
        ordenacion_rapida(v, 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.

```
//funcion para comprobar si un vector esta ordenado o no
bool compruebaOrdenado(std::vector<int> v)
{
    for(int i = 0; i < v.size()-1; i++)
    {
        if(v[i] > v[i+1])
            return false;
    }
    return true;
}

int main()
{
    std::vector<int> v;
    std::cout << "[Ordenacion rapida] Para un vector de 5 elementos, 120 combinaciones
        posibles" << std::endl;
    for(int i = 0; i < 5; i++)
        v.push_back(i);
    int ordenadas = 0;
    do{
        std::vector<int> aux = v;
```

```
        ordenacion_rapida(aux, 0, aux.size());
        if(compruebaOrdenado(aux))
            ordenadas++;
    }while(std::next_permutation(v.begin(), v.end()));
    std::cout << "Numero de permutaciones ordenadas: " << ordenadas << std::endl;
}
```

```
~/UC/DA/A/p3  main !12 ?8  g++ -o main pruebasCajaNegra.cpp
~/UC/DA/A/p3  main !12 ?8  ./main
[Fusion] Para un vector de 5 elementos, 120 combinaciones posibles
Numero de permutaciones ordenadas: 120
~/UC/DA/A/p3  main !12 ?8  g++ -o main pruebasCajaNegra.cpp
~/UC/DA/A/p3  main !12 ?8  ./main
[Ordenacion rapida] Para un vector de 5 elementos, 120 combinaciones posibles
Numero de permutaciones ordenadas: 120
~/UC/DA/A/p3  main !12 ?8  g++ -o main pruebasCajaNegra.cpp
~/UC/DA/A/p3  main !12 ?8  ./main
[Insercion] Para un vector de 5 elementos, 120 combinaciones posibles
Numero de permutaciones ordenadas: 120
~/UC/DA/A/p3  main !12 ?8  g++ -o main pruebasCajaNegra.cpp
~/UC/DA/A/p3  main !12 ?8  ./main
[Monticulo] Para un vector de 5 elementos, 120 combinaciones posibles
Numero de permutaciones ordenadas: 120
```

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.

   Partimos sabiendo que la suma del número de obstáculos y el número de defensas siempre será menor que el número de casillas del mapa.

   - **Sin preordenación:** la función rellenaVector() pertenece al $O(n)$, siendo n el número de casillas del mapa. El bucle while será de $O(n)$ en el peor de los casos (número de defensas = número de casillas factibles) y $O(defenses.size())$ en el mejor de los casos (todas las casillas seleccionadas sean factibles). La función factible tiene orden lineal (concretamente será igual al máximo entre el número de defensas y el número de obstáculos, llamémoslo x). El resto de operaciones que se hacen en la estrategia de colocación de defensas es de orden constante. Nos queda, por tanto $n + n * (n + x)$ en el peor de los casos y $n + defenses.size() * (n + x)$. Aplicando la regla del máximo, nos queda $O(n * n)$ en el peor de los casos y $O(defenses.size() * n)$ en el mejor de los casos.
   - **Ordenación por fusión:** la función rellenaVector() pertenece al $O(n)$, siendo n el número de casillas del mapa. El algoritmo de ordenacion por fusión pertence al orden $O(n * \log n)$ El bucle while será de $O(n)$ en el peor de los casos (número de defensas = número de casillas factibles) y $O(defenses.size())$ en el mejor de los casos (todas las casillas seleccionadas sean factibles). La función factible tiene orden lineal (concretamente será igual al máximo entre el número de defensas y el número de obstáculos, llamémoslo x). El resto de operaciones que se hacen en la estrategia de colocación de defensas es de orden constante. Nos queda, por tanto $n + n * (n * \log n + x)$ en el peor de los casos y $n + defenses.size() * (n * \log n + x)$. Aplicando la regla del máximo, nos queda $O(n^2 \log n)$ en el peor de los casos y $O(defenses.size() * n * \log n)$ en el mejor de los casos.
   - **Ordenación por ordenación rápida:** la función rellenaVector() pertenece al $O(n)$, siendo n el número de casillas del mapa. El algoritmo de ordenacion por ordenación rápida pertence al orden $O(n * \log n)$ El bucle while será de $O(n)$ en el peor de los casos (número de defensas = número de casillas factibles) y $O(defenses.size())$ en el mejor de los casos (todas las casillas seleccionadas sean factibles). La función factible tiene orden lineal (concretamente será igual al máximo entre el número de defensas y el número de obstáculos). El resto de operaciones que se hacen en la estrategia de colocación de defensas es de orden constante. Aplicando la regla del máximo, nos queda $O(n * \log n)$ en el peor de los casos.

- **Ordenación usando montículo:** la función rellenaVector() pertenece al $O(n)$, siendo n el número de casillas del mapa. std::make_heap() es de orden $O(n)$. El bucle while será de $O(n)$ en el peor de los casos (número de defensas = número de casillas factibles) y $O(defenses.size())$ en el mejor de los casos (todas las casillas seleccionadas sean factibles). std::pop_heap() tiene orden $O(\log n)$, luego el bucle tendrá $O(n * \log n)$ en el peor de los casos. La función factible tiene orden lineal (concretamente será igual al máximo entre el número de defensas y el número de obstáculos). El resto de operaciones que se hacen en la estrategia de colocación de defensas es de orden constante. Aplicando la regla del máximo, nos queda $O(n * \log n)$ en el peor de los casos.

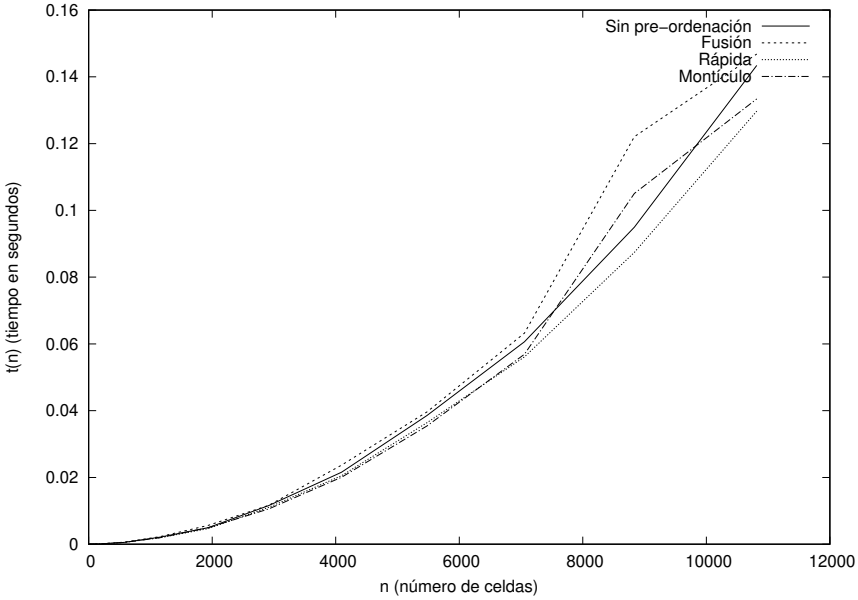
6. Incluya a continuación una gráfica con los resultados obtenidos. Utilice un esquema indirecto de medida (considere un error absoluto de valor 0.01 y un error relativo de valor 0.001). Es recomendable que diseñe y utilice su propio código para la medición de tiempos en lugar de usar la opción *-time-placeDefenses3* 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.

```
cronometro SO;
long int r1 = 0;
const double e_abs = 0.01, e_rel = 0.001;
SO.activar();
do {
    placeDefenses_SO(freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles
        , defenses);
    ++r1;
} while(SO.tiempo() < e_abs / e_rel + e_abs);
SO.parar();

cronometro OR;
long int r2 = 0;
OR.activar();
do {
    placeDefenses_OR(freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles
        , defenses);
    ++r2;
} while(OR.tiempo() < e_abs / e_rel + e_abs);
OR.parar();

cronometro F;
long int r3 = 0;
F.activar();
do {
    placeDefenses_F(freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles,
         defenses);
        ++r3;
} while(F.tiempo() < e_abs / e_rel + e_abs);
F.parar();

cronometro M;
long int r4 = 0;
M.activar();
do {
    placeDefenses_M(freeCells, nCellsWidth, nCellsHeight, mapWidth, mapHeight, obstacles,
         defenses);
        ++r4;
} while(M.tiempo() < e_abs / e_rel + e_abs);
M.parar();
std::cout << (nCellsWidth * nCellsHeight) << "\t" << SO.tiempo() / r1 << "\t" << F.tiempo
    () / r3 << "\t" << OR.tiempo() / r2 << "\t" << M.tiempo() / r4 << 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.