Práctica 3. Divide y vencerás

Sergio Rodríguez Vázquez
sergio.rodriguezvazquez@alum.uca.es
Teléfono: +634471641
NIF: 77171745Y

22 de diciembre de 2020

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

   Utilizo una estructura llamada C, que guarda la posición y el valor dado por defaultCellValue.

   En el primer caso, el algoritmo sin preordenación maneja una lista de C y va buscando la mejor posición en cada caso, comprobando que sea factible hasta que o bien termina la lista de C o ya no hay mas defensas que colocar.

   En los demás casos, los algoritmos manejan un vector de C.

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

```
std::vector<C> merge_sort(std::vector<C> &v, int i, int j)
{
    int n = j - i ;
    int k;
    if(n <= v.size())
    {
        Insertionsort(v, i, j);
    }
    else
    {
        k = i - 1 + n/2;
        merge_sort(v, i, k);
        merge_sort(v, k + 1, j);
        merge(v, i, k, j);
    }
    return v;
}

void Insertionsort(std::vector<C> &v, int i, int j)
{
    int key;
    for(i; i < j; i++)
    {
        key = v[i].value;
        j = i - 1;

        while(j >= 0 && v[j] > key)
        {
            v[j + 1] = v[j];
            j = j - 1;
        }
        v[j + 1].value = key;
    }

}

void merge(std::vector<C> &v, int i, int k, int j)
{
    int n, p, q;
    n = j - i + 1;
```

```
    p = i;
    q = k + 1;
    std::vector<C> w(n);
    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 - 1 + l] = w[l];
    }
}
```

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

```
std::vector<C> quicksort(std::vector<C> &v, int low, int high)
{
    if(low < high)
    {
        //pivot es el indice de particion
        int pivot = partition(v, low, high);

        //Separadamente ordenar elementos antes de la particion y despues de la particion
        quicksort(v, low, pivot - 1);
        quicksort(v, pivot + 1, high);
    }
    return v;
}

int partition(std::vector<C> &v, int low, int high)
{
    C pivot = v[high];
    int i = (low - 1);

    for(int j = low; j <= high - 1; j++)
    {
        if(v[j] <= pivot)
        {
            i++;
            std::swap(v[i], v[j]);
        }
    }
    std::swap(v[i + 1], v[high]);
    return (i + 1);
}
```

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 testalgorithm()
{
    std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    std::vector<int> w = {1, 2, 3, 4, 5, 6, 7, 8, 9};

    do{
        merge_sortint(v, 0, 8);
```

```
        if(!std::equal(v.begin(), v.end(), w.begin()))
        {
            std::cout << "v != w" << std::endl;
        }
    }while(std::next_permutation(v.begin(), v.end()));

    do{
        quicksortint(v, 0, 8);
        if(!std::equal(v.begin(), v.end(), w.begin()))
        {
            std::cout << "v != w" << std::endl;
        }
    }while(std::next_permutation(v.begin(), v.end()));
}
```

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.

   El orden de los distintos algoritmos es el siguiente:

   - Sin preordenación:
     $$O(n^2)$$
   - Fusión: O(n log n)
   - Ordenación rápida:
     $$O(n^2)$$
   - Montículo: O(n log n)

   Utilizaremos el algoritmo de Ordenación por Fusión.

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.

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.