Práctica 3. Divide y vencerás

David Luna Jurado
david.lunajurado@alum.uca.es
Teléfono: 663590384
NIF: 32098835N

17 de diciembre de 2022

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

   Para la representación del terreno de batalla utilizo una estructura casilla, la cual contiene su posición en el mapa representada por la fila y la columna en la que se encuentra, así como un campo valor que almacena el valor que se le dará a la defensa.

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

```
void ordenacionInsercion(int *v, int i, int j)
{
    int aux = 0;
    int k;
    for (int l = i + 1; l <= j; l++)
    {
        bool colocado = false;
        int m = l;
        for (k = l - 1; k >= i && !colocado; k--)
        {
            if (v[m] > v[k])
            {
                aux = v[k];
                v[k] = v[m];
                v[m] = aux;
                m--;
            }
        }
    }
}
void fusion(int *v, int *w, int i, int k, int j)
{
    int n = j - i + 1;
    int p = i;
    int q = k + 1;
    for (int l = 0; l < n; l++)
    {
        if (p <= k && (q > j || v[p] > v[q]))
        {
            w[l] = v[p];
            p = p + 1;
        }
        else
        {
            w[l] = v[q];
            q = q + 1;
        }
    }

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

```
    }
}
void ordenacionFusion(int *v, int *w, int i, int j)
{
    int n = j - i + 1;
    if (n <= 3)
    {
        ordenacionInsercion(v, i, j);
    }
    else
    {
        int k = i - 1 + n / 2;
        ordenacionFusion(v, w, i, k);
        ordenacionFusion(v, w, k + 1, j);
        fusion(v, w, i, k, j);
    }
}


void ordenacionRapida(int *v, int i, int j)
{
    int n = j - i + 1;
    if (n <= 3)
    {
        ordenacionInsercion(v, i, j);
    }
    else
    {
        int p = pivote(v, i, j);
        ordenacionRapida(v, i, p - 1);
        ordenacionRapida(v, p + 1, j);
    }
}
```

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

```
void ordenacionInsercion(int *v, int i, int j)
{
    int aux = 0;
    int k;
    for (int l = i + 1; l <= j; l++)
    {
        bool colocado = false;
        int m = l;
        for (k = l - 1; k >= i && !colocado; k--)
        {
            if (v[m] > v[k])
            {
                aux = v[k];
                v[k] = v[m];
                v[m] = aux;
                m--;
            }
        }
    }
}

int pivote(int *v, int i, int j)
{
    int p = i;
    int x = v[i];
    int aux;
    for (int k = i + 1; k <= j; k++)
    {
        if (v[k] >= x)
        {
            p = p + 1;
            aux = v[p];
            v[p] = v[k];
```

```
            v[k] = aux;
        }
    }
    v[i] = v[p];
    v[p] = x;
    return p;
}
```

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.

```
// La funcion prueba todas las permutaciones posibles que existen a partir de un vector
    ordenada ascendentemente, y las ordena de una en una para comprobar que funciona la
    funcion de ordenacion pasada por parametro
void cajaNegra(int *v, int n, void(*ordenacion)(int*,int*,int,int))
{
    int copia[n];
    int w[n];
    int permutaciones[n];
    memcpy(permutaciones, v, n * sizeof(int));
    do
    {
        memcpy(copia, permutaciones, n * sizeof(int));
        ordenacion(copia, w,0, n - 1);
        std::cout << "Vector Original\n";
        display(permutaciones, n); //La funcion display muestra un vector por pantalla
        std::cout << "Ordenado\n";
        display(copia, n);
    } while (std::next_permutation(
        permutaciones, permutaciones + n));
}

int main()
{
    int n = 5;
    int numeros[n] = {1, 2, 3, 4, 5};

    cajaNegra(numeros,n,ordenacionRapida);
    cajaNegra(numeros,n,ordenacionFusion);

    std::cout << std::endl;

    return 0;
}
```

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.

   Siendo n el número total de celdas de un mapa y d el número de defensas que tenemos que colocar:

   En un algoritmo voraz las operación crítica es la operación de selección.

   Si no preordenamos las defensas, la función de selección será de orden $O(n)$, ya que la función debe recorrer toda la lista de casillas para determinar cual es la mejor.

   Esa función será ejecutada un total de d veces en el mejor caso, ya que todas las defensas se podrían colocar en la casilla que se seleccionase. En el peor caso se ejecutará un total de n veces, si se acaban seleccionando todas las casillas.

   En cada iteración se elimina una casilla de la lista por lo que cada vez el tiempo de ejecución de la función de selección disminuye en una comparación. Por lo tanto:

$$t(n) = t(n-1) + 1 = \sum_{i=1}^{n} n - i = \frac{n^2 - n}{2} \in O(n^2)$$

En el caso de que se ordenen la lista de casilla previamente obtenemos un tiempo de selección de $O(1)$, que al repetirse como máximo n veces tendríamos un tiempo de $O(n)$ para todos los algoritmos en los que se preordene la lista de casillas. Por lo tanto, para decantarnos por uno de ellos tendremos que analizar el orden de complejidad de los propios algoritmos de ordenación.

Suponiendo un mapa cuadrado, el algoritmo de ordenación por fusión siempre podrá dividir el vector de casillas de forma equitativa, por lo que, tal y como se analizó en clase obtenemos que el orden del algoritmo en el peor caso es de:

Si $n > n_0$, donde en este caso $n_0 = 3$

$$t(n) = 2t(\frac{n}{2}) + n$$

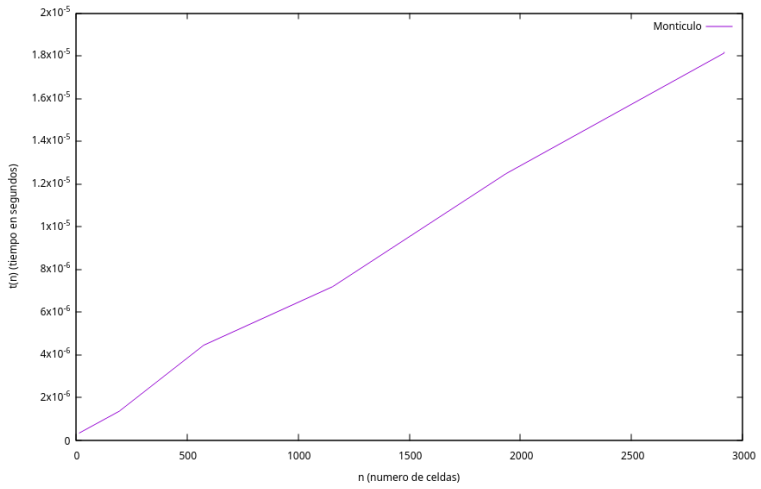
$$t(n) \in O(n \log n)$$

Analizando el algoritmo de ordenacion rápida en el peor caso, (el pivote queda en alguno de los extremos) obtenemos un orden de $O(n^2)$ pero en el caso promedio y en el mejor caso obtenemos un orden perteneciente a $O(n \log n)$

En caso de usar un montículo la función de selección no es $O(1)$ si no de $O(\log n)$. La inserción de elementos en un montículo es de orden $O(logn)$ en el peor caso, como hay que insertar n elementos acaba siendo $O(n \log n)$. Y como en el peor caso se extraen n elementos del montículo, pero como el arbol se va haciendo más pequeño en cada iteración el orden es $O(\log n!)$

Como en el caso peor en la inserción de un montículo tomaremos ese como nuestro algoritmo elegido.

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.