PRÁCTICA 1

Complejidad Computacional
2B

Diego Viñals Lage

Carlos Vega Colado

Tabla de contenido

Introducción

Esta práctica consiste en ordenar unos arrays con unos algoritmos de ordenación. Para cada array hemos escogido unos algoritmos acordes con el tamaño del array y lo parcialmente ordenado que estaba inicialmente.

Array 1

int [] array1={ 20, 9, 23, 13, 6, 18, 25, 17, 4, 8, 15, 22, 16, 3, 24, 21, 14, 12, 1, 5, 7, 11, 19, 2, 10}

Con este array hemos escogido el algoritmo de SelectionSort, ya que es muy útil para arrays pequeños y simples, el caso de este array. Este array al tener elementos muy cercanos a sí mismos es fácil de ordenar siguiendo este algoritmo.

```
static void sort(int[] arr)
{
    int n = arr.Length;

    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n - 1; i++)
    {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first
        // element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Este algoritmo selecciona el elemento más pequeño y lo pone al principio, y continua con el resto de los elementos. Tiene complejidad n^2 y no es estable. Si quisiéramos que sea estable habría que usar BubbleSort.

Array 2

int [] array2={ 1, 2, 3, 4, 20, 6, 7, 8, 24, 10, 11, 12, 14, 13, 15, 16, 17, 18, 19, 5, 21, 22, 23,9, 25}

Este array esta parcialmente ordenado, por lo que en vez de usar SelectionSort hemos preferido usar insertionSort, ya que va cambiando los elementos por parejas, es decir, va comparando parejas de elementos, y si esta parcialmente ordenado es muy útil porque en muchas iteraciones no tiene que cambiar ningún elemento.

```
// Function to sort array
// using insertion sort
void sort(int[] arr)
{
    int n = arr.Length;
    for (int i = 1; i < n; ++i)
    {
        int key = arr[i];
        int j = i - 1;

        // Move elements of arr[0..i-1],
        // that are greater than key,
        // to one position ahead of
        // their current position
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Este algoritmo es estable y tiene complejidad n^2 y es estable. Si por algún motivo queremos que no sea estable podríamos usar SelectionSort.

Array 3

int [] array3= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25}

Como en el array 2 hemos escogido utilizar insertionSort, ya que podemos ver que este array ya está ordenado. Este algoritmo comprobaría por pares los elementos y al estar ordenados no cambiaría ningún elemento.

```
// Function to sort array
// using insertion sort
void sort(int[] arr)
{
    int n = arr.Length;
    for (int i = 1; i < n; ++i)
    {
        int key = arr[i];
        int j = i - 1;

        // Move elements of arr[0..i-1],
        // that are greater than key,
        // to one position ahead of
        // their current position
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Este algoritmo es estable y tiene complejidad n^2

Array 4

Int []array4={ 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2,1}

Podemos ver que este array esta del revés, ordenados de mayor a menor cuando lo queremos de menor a mayor. Hemos elegido usar InsertionSort, ya que tiene tamaño pequeño.

```
// Function to sort array
// using insertion sort
void sort(int[] arr)
{
    int n = arr.Length;
    for (int i = 1; i < n; ++i)
    {
        int key = arr[i];
        int j = i - 1;

        // Move elements of arr[0..i-1],
        // that are greater than key,
        // to one position ahead of
        // their current position
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

La complejidad de este algoritmo seria n^2

Array 5

int []array5={ 3, 8, 9, 6, 7, 5, 2, 3, 1, 4, 9, 7, 8, 10}

Este array es el más pequeño de todos, por lo que hemos preferido usar SelectionSort. Este algoritmo es el mejor para arrays pequeños y simples, como este.

```
static void sort(int[] arr)
{
    int n = arr.Length;

    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n - 1; i++)
    {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first
        // element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Ya hemos explicado arriba como funciona. Su complejidad es n^2

Array 6

double[] array6= { 6.9, 7.7, 5.1, 7.5, 7.8, 5.5, 7.3, 5.8,7.9, 6.1, 5.2, 6.4, 6.7,7.4, 6.5, 6.8,7.1, 5.9, 7.0, 6.0,5.7, 5.4,7.2, 7.6,5.0, 5.3, 6.6, 8.0,5.6,6.2, 6.3 };

Para este hemos elegido utilizar SelectionSort, por lo mismo que hemos visto antes, es simple y pequeño.

```
static void sort(int[] arr)
{
    int n = arr.Length;

    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n - 1; i++)
    {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first
        // element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Su complejidad es n^2

Array 7

int [] array7={ 130, 690, 1888, 2579, 1,9179, 6313,748, 1514, 8103, 4998,7211, 5144, 9127, 5, 777, 1113, 19, 2, 9304,9999, 981, 394, 805}

Este array es pequeño, pero tiene elementos muy grandes y pequeños. Para este hemos preferido usar SelectionSort, ya que no nos afecta que los elementos estén muy dispersos. El array es pequeño y simpe. Su complejidad es n^2

```
static void sort(int[] arr)
{
    int n = arr.Length;

    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n - 1; i++)
    {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first
        // element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Array 8

string [] array8={ “J”, ”G” , “M”, “A”, “N”, ”E” , “Z”, “T”, “B”, “O”, “D”,”C” }

Este array en vez de ser números tiene como elementos una serie de strings, pero no nos afecta mucho porque los string tienen un numero ASCII asociado por lo que se puede ordenar exactamente igual como si fueran números.

Este lo queremos hacer con SelectionSort, ya que es simple, pequeño, y no esta parcialmente ordenado.

```
static void sort(int[] arr)
{
    int n = arr.Length;

    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n - 1; i++)
    {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first
        // element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Su complejidad es de n^2

Array 9

Este array es un array muy grande, por lo que no lo ponemos en la memoria. Tiene elementos muy grandes y pequeños.

Usamos QuickSort porque este algoritmo está diseñado para arrays grandes. Usa el paradigma divide y vencerás, por lo que va dividiendo el array en 2 recursivamente, hasta que ordene todos los elementos.

Este algoritmo no es estable, por lo que es posible que sea mejor usar MergeSort ya que este sí que es estable.

La complejidad de este algoritmo es de $n\log(n)$

```
// A utility function to swap two elements
static void swap(int[] arr, int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

/* This function takes last element as pivot, places
     the pivot element at its correct position in sorted
     array, and places all smaller (smaller than pivot)
     to left of pivot and all greater elements to right
     of pivot */
static int partition(int[] arr, int low, int high)
{

    // pivot
    int pivot = arr[high];

    // Index of smaller element and
    // indicates the right position
    // of pivot found so far
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {

        // If current element is smaller
        // than the pivot
        if (arr[j] < pivot)
        {

            // Increment index of
            // smaller element
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, high);
    return (i + 1);
}
```

```java
/* The main function that implements QuickSort
            arr[] --> Array to be sorted,
            low --> Starting index,
            high --> Ending index
   */
static void quickSort(int[] arr, int low, int high)
{
    if (low < high)
    {

        // pi is partitioning index, arr[p]
        // is now at right place
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```