Ayudantía 2

Insertion Sort
Sala de Ayuda T0

Insertion Sort

Insertion Sort

Sala de Ayuda T0

1. Tenemos una secuencia desordenada

2. Tomar el primer dato 'x' de la secuencia

3. Insertar 'x' en los elementos anteriores de manera que quede ordenado

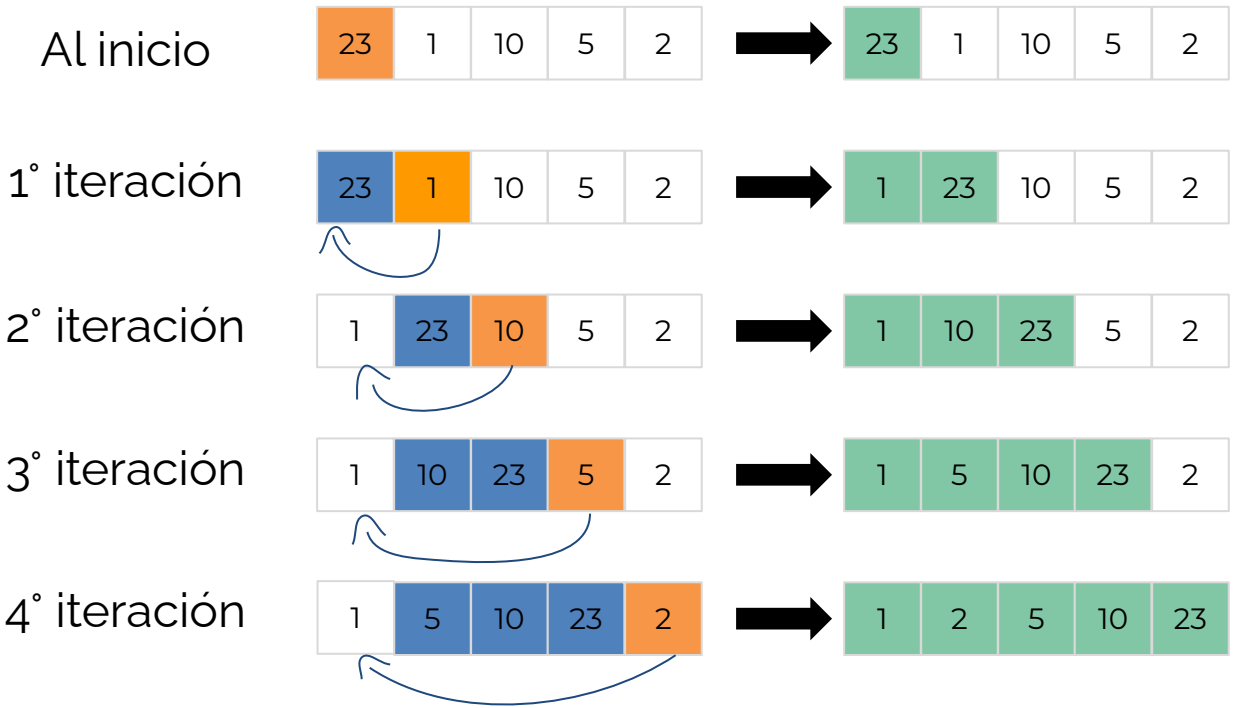
4. Avanzar en la secuencia

5. Si aún queda secuencia, *volver al paso 2*

Insertion Sort

Ejemplo

Se tiene un arreglo, **A = [23, 1, 10, 5, 2]** y queremos ordenarlo



```c
/* Function to sort array using insertion sort */
void insertionSort(int arr[], int n)
{
    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;
    }
}
…
```

```c
/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        printf("%d ", arr[i]);
    printf("\n");
}

int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

Source Code

Insertion Sort en C

```c
/* Function to sort array using insertion sort */
void insertionSort(int arr[], int n)
{
    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;
    }
}
…
```

```c
/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        printf("%d ", arr[i]);
    printf("\n");
}

int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

Source Code

Insertion Sort en C

Problema 2018-2-I1-P3-b)

Calcula **cuántas comparaciones** entre elementos hace el siguiente algoritmo ***shellSort( )*** para ordenar el arreglo
a =[11, 10, 9, 8, 7, 6, 5, 4, 3, 2 , 1] .
Muestra que entiendes cómo funciona shellSort( ); en particular, ¿qué relación tiene con insertionSort( )?

```
shellSort(a):
    gaps[] = {5,3,1}
    t = 0
    while t < 3:
        gap = gaps[t]
        j = gap
        while j < a.length]:
            tmp = a[j]
            k = j
            while k >= gap and tmp < a[k-gap]:
                a[k] = a[k-gap]
                k = k-gap
            a[k] = tmp
            j = j+1
    t = t+1
```

```
shellSort(a):
    gaps[] = {5,3,1}
    t = 0
    while t < 3:
        gap = gaps[t]
        j = gap
        while j < a.length]:
            tmp = a[j]
            k = j
            while k >= gap and tmp < a[k-gap]:
                a[k] = a[k-gap]
                k = k-gap
            a[k] = tmp
            j = j+1
    t = t+1
```

Solución: Cómo funciona ShellSort

El algoritmo **ShellSort** es una versión optimizada del **InsertionSort**, donde en lugar de comparar y ordenar elementos adyacentes, se compara con elementos separados por una distancia determinada por una secuencia de **gaps**.

En este caso, el algoritmo usa: *gaps = {5, 3, 1}*

Por ende, en cada iteración con un gap se aplica InsertionSort en los elementos separados por esa distancia

```
shellSort(a):
    gaps[] = {5,3,1}
    t = 0
    while t < 3:
        gap = gaps[t]
        j = gap
        while j < a.length]:
            tmp = a[j]
            k = j
            while k >= gap and tmp < a[k-gap]:
                a[k] = a[k-gap]
                k = k-gap
            a[k] = tmp
            j = j+1
    t = t+1
```

```
shellSort(a):
    gaps[] = {5,3,1}
    t = 0
    while t < 3:
        gap = gaps[t]
        j = gap
        while j < a.length]:
            tmp = a[j]
            k = j
            while k >= gap and tmp < a[k-gap]:
                a[k] = a[k-gap]
                k = k-gap
            a[k] = tmp
            j = j+1
    t = t+1
```

Solución: Seguimiento Comparaciones

Importante que notemos que las comparaciones se realizan en la línea:

```
while k >= gap and tmp < a[k-gap]:
```

Esta comparación se ejecuta *cada vez* que un elemento es comparado con otro en el proceso de ordenamiento

```
shellSort(a):
    gaps[] = {5,3,1}
    t = 0
    while t < 3:
        gap = gaps[t]
        j = gap
        while j < a.length]:
            tmp = a[j]
            k = j
            while k >= gap and tmp < a[k-gap]:
                a[k] = a[k-gap]
                k = k-gap
            a[k] = tmp
            j = j+1
    t = t+1
```

```
shellSort(a):
    gaps[] = {5,3,1}
    t = 0
    while t < 3:
        gap = gaps[t]
        j = gap
        while j < a.length]:
            tmp = a[j]
            k = j
            while k >= gap and tmp < a[k-gap]:
                a[k] = a[k-gap]
                k = k-gap
            a[k] = tmp
            j = j+1
    t = t+1
```

Solución: Seguimiento Comparaciones

gap = 5

- Se realiza **InsertionSort** en elementos separados por **5 posiciones en el arreglo**
- Hay 0 comparaciones *false* (no hay que hacer swaps)
- Hay 7 comparaciones *true* (sí se debe hacer swap)
- En total: **7 comparaciones**

```
shellSort(a):
    gaps[] = {5,3,1}
    t = 0
    while t < 3:
        gap = gaps[t]
        j = gap
        while j < a.length]:
            tmp = a[j]
            k = j
            while k >= gap and tmp < a[k-gap]:
                a[k] = a[k-gap]
                k = k-gap
            a[k] = tmp
            j = j+1
    t = t+1
```

Solución: Seguimiento Comparaciones

gap = 3

- Se realiza **InsertionSort** en elementos separados por **3 posiciones en el arreglo**
- Hay 7 comparaciones *false* (no hay que hacer swaps)
- Hay 2 comparaciones *true* (sí se debe hacer swap)
- En total: **9 comparaciones**

```
shellSort(a):
    gaps[] = {5,3,1}
    t = 0
    while t < 3:
        gap = gaps[t]
        j = gap
        while j < a.length]:
            tmp = a[j]
            k = j
            while k >= gap and tmp < a[k-gap]:
                a[k] = a[k-gap]
                k = k-gap
            a[k] = tmp
            j = j+1
    t = t+1
```

Solución: Seguimiento Comparaciones

gap = 1

- Se realiza **InsertionSort** en elementos separados por **1 posiciones en el arreglo**
- Hay 10 comparaciones *false* (no hay que hacer swaps)
- Hay 2 comparaciones *true* (sí se debe hacer swap)
- En total: **12 comparaciones**

```
shellSort(a):
    gaps[] = {5,3,1}
    t = 0
    while t < 3:
        gap = gaps[t]
        j = gap
        while j < a.length]:
            tmp = a[j]
            k = j
            while k >= gap and tmp < a[k-gap]:
                a[k] = a[k-gap]
                k = k-gap
            a[k] = tmp
            j = j+1
    t = t+1
```

Solución: Seguimiento Comparaciones

All gaps

- **En total**: **28 comparaciones**!
- 11 son true y 17 son false

```
shellSort(a):
    gaps[] = {5,3,1}
    t = 0
    while t < 3:
        gap = gaps[t]
        j = gap
        while j < a.length]:
            tmp = a[j]
            k = j
            while k >= gap and tmp < a[k-gap]:
                a[k] = a[k-gap]
                k = k-gap
            a[k] = tmp
            j = j+1
    t = t+1
```

```
shellSort(a):
    gaps[] = {5,3,1}
    t = 0
    while t < 3:
        gap = gaps[t]
        j = gap
        while j < a.length]:
            tmp = a[j]
            k = j
            while k >= gap and tmp < a[k-gap]:
                a[k] = a[k-gap]
                k = k-gap
            a[k] = tmp
            j = j+1
    t = t+1
```

Solución: Pauta

Notemos que las comparaciones entre elementos de a se dan sólo en la comparación **tmp < a[k-gap]**; el algoritmo realiza 11 de estas comparaciones con resultado true y otras 17 con resultado false; en total, 28.

Primero, realiza insertionSort entre elementos que están a distancia 5 entre ellos (según las posiciones que ocupan en a, no en cuanto a sus valores): el 6 con respecto al 11, el 5 c/r al 10, el 4 c/r al 9, el 3 c/r al 8, el 2 c/r al 7, el 1 c/r al 11, y el 1 c/r al 6.

Luego, realiza insertionSort entre elementos que están a distancia 3 (nuevamente, según sus posiciones en el arreglo): el 2 c/r al 5 y el 7 c/r al 10.

Finalmente, realiza insertionSort entre elementos que están a distancia 1: el 3 c/r al 4 y el 8 c/r al 9; estos son los dos únicos pares de valores que aún están "desordenados" al finalizar el paso anterior.

Sala de Ayuda T0

Insertion Sort

Sala de Ayuda T0