Universidad Distrital Francisco José De Caldas

Análisis: Insertion sort - Heap sort

Davidson Esfleider Sanchez Gordillo - 20231020183
Diego Felipe Diaz Roa - 20201020147
Dania Lizeth Guzmán Triviño - 20221020061
Joshoa Alarcon Sanchez - 20221020013

Profesor
Simar Enrique Herrera Jimenez

Bogotá D.C.
2025

Insertion sort

idea general:

El insertion sort recorre el arreglo elemento por elemento. Los nuevos elementos se insertan en la posición correcta respecto a los anteriores elementos, desplazándolos si es necesario.

Conteo de pasos:

Para cada posición i, desde 2 hasta n:
compara con los anteriores y los desplaza hasta encontrar su lugar.
En el peor caso:
El primer elemento se compara 1 vez, el segundo 2, el tercero 3, así sucesivamente hasta n - 1.
Total de comparaciones $\approx 1 + 2 + 3 + \ldots + (n-1) = n(n-1)/2$

Complejidad:

- Peor caso: O(n²)
- Mejor caso: O(n)
- Caso promedio: O(n log n)

Heap sort

idea general:

En el Heap sort se construye un montículo con los datos. Repetidamente se intercambia el mayor con el último elemento, se reduce el tamaño del heap y se reacomoda.

Conteo de pasos:

- construcción del heap:
    - Se aplica heapify de abajo hacia arriba.
    - El costo total no es de *n log n* como parece, sino *n*, porque los nodos más profundos requieren menos operaciones.

- Extracciones sucesivas:
    - Cada extracción cuesta *log n,* esto se debe al heapify.
    - Se hacen *n* extracciones.
    - Total: n*O(log n) = O(n log n)

Complejidad:

- Peor caso: O(n log n)
- Mejor caso: O(n log n)
- Caso promedio: O(n log n)

Comparación

	Insertion sort	**Heap sort**
complejidad	mejor: O(n) Peor: $O(n^2)$	Siempre: O(n log n)
Datos pequeños	Muy eficiente cuando trabaja con datos pequeños.	Funciona, pero el proceso de Overhead lo hace más lento.
Datos grandes	Suele ser poco eficiente con grandes cantidades de datos.	En este caso, suele ser mucho más eficiente.
Orden previo	Aprovecha si varios de los datos se encuentran ordenados.	No hace provecho del orden previo.
Estabilidad	Sí, mantiene el orden relativo de los iguales.	No es estable.
Aplicaciones	Es ideal para listas pequeñas o que tengan un orden previo.	Mejor en listas grandes con datos que se encuentren en desorden.

Comparación visual

Insertion sort

Insertion.py > ...

```python
import random

array = random.sample(range(2000),1000)
n = 0

def insertion_sort(arr):
    global n

    for i in range(1, len(arr)):
        # Llave y primer dato a comparar
        key = arr[i]
        j = i - 1
        n += 2

        # mover datos a la derecha si son mayores a la clave hasta insertar la clave
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
            n += 4

        arr[j + 1] = key
        n += 1

insertion_sort(array)
print(array, n)

```

Heap sort

heap.py > ...

```python
import random

array = random.sample(range(2000),1000)
test = [9, 4, 3, 8, 10, 2, 5]
n = 0 # pasos

def heapify(arr, size, index):
    global n

    left = 2 * index + 1
    right = 2 * index + 2
    largest = index
    n += 3

    if (left < size) and (arr[left] > arr[largest]):
        largest = left
        n += 1

    if (right < size) and (arr[right] > arr[largest]):
        largest = right
        n += 1

    if largest != index:
        arr[index], arr[largest] = arr[largest], arr[index]

        heapify(arr, size, largest)
        n += 2

    n += 5
```

```python

def heap_sort(arr):
    global n

    size = len(arr)
    n += 1

    #recorrer desde abajo hacia arriba todo el arbol para mover los mayores (alistar array)
    for i in range(size//2 - 1, -1, -1):
        heapify(arr, size, i)
        n += 1

    # Ordenar array
    for i in range(size - 1, 0, -1):
        #raiz del todo el arbol se pone al final (ordenar dato mayor)
        arr[0], arr[i] = arr[i], arr[0]

        #volver a poner el mayor en la raiz
        heapify(arr, i, 0)
        n += 2


heap_sort(array)
print(array, n)
```

Tabla con datos de prueba:

Datos	***50***	***100***	***500***	***1000***
Insertion	2787	9165	246905	996849
Heap	2855	6940	47933	106730

Según los resultados obtenidos a través de diferentes pruebas realizadas, se puede observar que a pesar de que en casos constantes se supone que ambos algoritmos tienen un nivel de complejidad O(*n log n*), esta misma (la complejidad) varía dependiendo la cantidad de datos.

En el caso del insertion, el cual presenta la mayor variación, entre menor la cantidad de datos su eficiencia es mayor. No obstante, entre más sea la cantidad de información que deba ordenar, su complejidad se acerca a n^2. Esto se debe a la cantidad de veces que tiene que desplazar los elementos de array cada vez que va a ordenar un dato.

Por parte de Heap, este mantiene su complejidad O(*n log n*), siendo más eficiente y consistente en cualquier escenario. Aunque, como se puede observar con una menor cantidad de datos, no es el más rápido en la práctica.

Conclusión

Entre los algoritmos observados a través de las clases, el Heap sort —en conjunto al Quick sort—, son de los algoritmos más efectivos. Esto se debe a que en todos los casos suelen usar la misma cantidad de pasos, causado por el hecho de que su complejidad *n log n* es muy

eficaz incluso comparados con otros que también comparten esta complejidad; además de mantener su eficacia tanto con grupos de datos pequeños como muy grandes.