Análisis de algoritmos : Taller 1

Carlos Escobar [1] Fabián Rojas[1] Julián Tarazona[1]

[1]Pontificia Universidad Javeriana
{escobartc, fabian-rojasm, jdtarazona}@javeriana.edu.co
February 17, 2022

1 Resumen

En este documento se presenta el problema del ordenamiento de elementos y su solución a través del análisis, diseño y futura implementación del algoritmo de TimSort, el cual es una combinación de los algoritmos merge sort y binary insertion sort, que es a su vez una mezcla de insertion sort y binary search. También se basa en el patrón dividir y vencer, además de ser utilizado como el algoritmo de ordenamiento nativo para el lenguaje Python.

2 Análisis y diseño

2.1 Análisis

El ejercicio parte desde la necesidad de ordenar una secuencia de elementos utilizando el algoritmo de ordenamiento TimSort (algoritmo nativo de Python), el cual sigue el principio de dividir y vencer. Para esto, la secuencia de números a ordenar debe cumplir las siguientes condiciones:

- La secuencia de números debe ser entera.
- La secuencia de números debe hacer parte del conjunto $\mathbb{T}$ de elementos ordenables.

Por otro lado, en lo que respecta al algoritmo Timsort, este es de gran utilidad y es eficiente en la realidad, pues sigue la premisa que dice que en el arreglo de los datos a ordenar existen ya subsecuencias ordenadas de los mismos datos.

Timsort tiene una forma de comportarse a acuerdo a la cantidad de elementos que tenga el arreglo, es decir si $|\mathbb{S}| \leq 64$, entonces se ejecutará el método de *Insertion Sort* por su eficiencia en ordenar con una cantidad pequeña de elementos.

Si por el contrario, $|\mathbb{S}| > 64$ funciona de una manera "sencilla" a través de la combinación de algoritmos ya existentes, partiendo de la división de un arreglo en secciones que llevan el nombre de *runs* (ejecuciones), estas no son más que las subsecuencias decrecientes o no decrecientes más larga. Posteriormente se hace una clasificación de esas ejecuciones a través del método de ordenamiento *Insertion Sort* y finalmente se combinan estas soluciones haciendo uso del método por ordenamiento conocido como *Merge Sort*.

2.2 Diseño

El diseño se compone de la definición de las siguientes entradas y salidas:

1. **Entradas:** Una secuencia de números tal que $S = \langle s_1, s_2, \cdots, s_n \rangle = \langle s_i \in \mathbb{T} \wedge 1 \leq i \rangle$ Donde $\mathbb{T}$ son el conjunto de elementos que pueden ser ordenables, n es la cardinaliad de la secuencia e i es el indice de cada elemento. En dicho conjunto debe estar definida la relación de orden parcial $\leq$.
2. **Salidas:** El algoritmo retornará una permutación S' de la secuencia de números S de forma que $S' = \langle s'_i \in S \wedge s'_{i-1} \leq s'_i \forall 1 < i \wedge 1 \leq i \leq n \rangle$.

3 Algoritmos

3.1 Pseudo-Código

Algorithm 1 Timsort.

```
1: procedure TIMSORT(S,b,e)
2:     if e − b < 64 then
3:         InsertionSort(S)
4:     else
5:         q ← ⌊(b + e) ÷ 2⌋
6:         Timsort(S, b, q)
7:         Timsort(S, q + 1, e)
8:         Merge(s, b, q, e)
9:     end if
10: end procedure
```

Algorithm 2 Merge.

```
1: procedure MERGE(S,b,q,e)
2:     n1 ← q − b + 1
3:     n2 ← e − q
4:     let L[1, n1 + 1] and R[1, n2 + 1]
5:     for i ← 0 to n1 do
6:         L[i] ← S[b + i − 1]
7:     end for
8:     for i ← 0 to n2 docn
9:         R[i] ← S[q + i]
10:    end for
11:    L[n1 + 1] ← ∞ ∧ R[n2 + 1] ← ∞
12:    i ← 1 ∧ j ← 1
13:    for k ← b to e do
14:        if L[i] < R[j] then
15:            S[k] ← L[i]
16:            i ← i + 1
17:        else
18:            S[k] ← R[j]
19:            j ← j + 1
20:        end if
21:    end for
22: end procedure
```

Algorithm 3 InsertionSort

```
1: procedure INSERTIONSORT(S)
2:     for j ← 2 to |S| do
3:         k ← S[j]
4:         i ← j − 1
5:         while 0 < i ∧ k < S[i] do
6:             S[i + 1] ← S[i]
7:             i ← i − 1
8:         end while
9:         S[i + 1] ← k
10:    end for
11: end procedure
```

3.2 Complejidad

El algoritmo Timsort identifica ejecuciones, que son subsecuencias ascendentes y descendentes, usa además el Binary Insertion Sort dependiendo de la cardinalidad de la secuencia, un algoritmo que es más rápido que el Insertion Sort, luego mezcla las subsecuencias identificadas con el Merge Sort. Este algoritmo se comporta con normalidad, pues la complejidad no varía y por lo tanto es un algoritmo de ordenación estable, por lo que, tras inspeccionar el algoritmo con el teorema maestro, se determinó que este tiene un orden de complejidad **O(n(log(n))**.

3.3 Invariante

Invariante de TimSort:

1. **Inicio:** El resultado de comparar el tamaño del arreglo original con el minRun.
2. **Avance:** Se valida que el valor del tamaño de cada subarreglo (que cambia en cada itración) sea menor al minRun, se calcula el nuevo punto de partida para el subarreglo.
3. **Terminación:** El tamaño del arreglo resulta ser mayor al tamaño del minRun.

Invariante de Merge:

1. **Inicio:**El arreglo S contiene un elemento que está trivialmente ordenado.
2. **Avance:** Se asume que el k-ésimo elemento de los subarreglos L y R es el más pequeño y ya se encuentra ordenado en S.
3. **Terminación:** Todos los elementos fueron reinsertados en el arreglo S y se encuentran ordenados.

Invariante de InsertionSort:

1. **Inicio:** El arreglo de la forma A[0,n] se recorrera en ese limite.
2. **Avance:** En la i-ésima iteración el elemento actual se compara con su predecesor, moviendo hacia la derecha los elementos que sean mayores al actual.
3. **Terminación:** Todo lo que se encuentra a la izquierda de la secuencia de números se encuentra ordeando en la n-ésima iteración.