Universidad de los Andes > FACULTAD DE INGENIERÍA Y CIENCIAS APLICADAS

Estructuras de Datos y Algoritmos (EDA)
Tarea 2: Ordenamiento de Orden Lineal

Prof: José M. Saavedra & David Miranda

Fecha de entrega: 30 de septiembre de 2025

1. Objetivo

Implementar un método de ordenación lineal implementando ADTs basados en listas enlazadas y comparar la eficiencia con respecto a métodos de ordenación genéricos de orden $O(n \log n)$.

2. Descripción

La agencia de correos necesita diseñar un algoritmo para ordenar los paquetes de acuerdo con el **código postal**. En este caso, el código postal se compone de 4 dígitos y dos letras. A continuación se muestran algunos ejemplos de códigos.

```
2553FI
7561PE
7336IU
8311BH
4615GG
4444YI
1753KV
7445KA
3171PK
2323FO
```

Tú, como especialista en algoritmos, sabes que RadixSort es una excelente alternativa para ordenar los códigos postales. Por lo tanto, en esta tarea deberás implementar RadixSort usando CountingSort para ordenar cada columna de valores de los códigos.

El algoritmo descrito en Alg. 1 utiliza un arreglo L para guardar listas que hacen referencia a los códigos. Así, tú deberás crear una Lista Enlazada que te permita guardar los códigos que comparten un valor con respecto a una de sus columnas. Por cuestiones de eficiencia es preferible guardar las posiciones de los códigos en vez del contenido mismo.

Considerando el algoritmo anterior, ahora podemos definir RadixSort, tal como se indica en el Algoritmo 2.

Para demostrar la eficiencia de RadixSort, tendrás que comparar su desempeño contra los algoritmos QuickSort y MergeSort.

Algorithm 1 Algoritmo CountingSort (CS)

Require: $A[1 \cdots n]$: arreglo con n códigos postales
Require: n: número de códigos postales
Require: p: posición de la columna a ordenar $(1, \cdots, 6)$
Require: M: tamaño del rango de valores a ordenar.

```
L ← crear arreglo de Listas Enlazadas de tamaño M
for c ∈ A do
    agregar c en L[c.getValue[p] + 1]
end for
borrar contenido de A
k ← 1
for i ← 1 ··· M do
    for e ∈ L[i] do
        A[k] ← e
        k ← k + 1
    end for
end for
```

Algorithm 2 Algoritmo RadixSort (CS)

Require: $A[1 \cdots n]$: arreglo con n códigos postales
Require: n: número de códigos postales

```
CS(A, n, 6, 26)
CS(A, n, 5, 26)
for i ← 4 ··· 1 do
    CS(A, n, i, 10)
end for
```

2.1. Conjuntos de Datos

En esta tarea usarás 3 conjuntos de datos con diversa cantidad de códigos:

- *codes_500K*: contiene 500.000 códigos.
- *codes_1M*: contiene un millón de códigos.
- *codes_10M*: contiene diez millones de ćodigos.

Puedes descargas los archivos de datos desde `https://www.dropbox.com/scl/fi/ju7zgg4xellugvgfnuf74/poscodes.zip?rlkey=gl8ju4rxiv2yp3tud1pgnydw3&st=0xddcjk6`.

2.2. Resumen

1. Implementar una lista enlazada que represente la lista de códigos que se guarda en cada celda de L. Por fines de optimización, guarda la posición del código en lugar de su valor. Es crítico que cada inserción en la lista enlazada cueste $O(1)$, para garantizar tiempo $O(n)$ en *CountingSort*.
2. Implementar *RadixSort* usando *CountingSort* como base. Para esto debes seguir los algoritmos propuestos. Sin embargo, siéntete libre de realizar algunas modificaciones con fines de hacerlos más eficientes.
3. Implementar QuickSort y MergeSort de modo que orden a lista de códigos.
4. Calcular el tiempo de procesamiento de cada método de ordenación sobre 3 conjuntos de datos y llenar la tabla 1. Para tener un resultado más confiable, corre 5 veces cada experimento y reportar el promedio junto a su desviación estándar con respecto a cada celda de la tabla indicada. Por ejemplo, corre 5 veces el experimento de Radix para 500K y reporta el promedio y su desviación

de la siguiente manera: ($\mu_{r1} \pm \sigma_{r1}$), donde μ_{r1} es el promedio y σ_{r1} es la desviación. Sigue el mismo enfoque para el resto de celdas.

5. Todas las implementaciones deben ser desarrolladas sobre el código base disponible en `https://github.com/jmsaavedrar/eda_cpp/tree/main/poscodes`.

Tamaño	**Radix**	**Quick**	**Merge**
500K	$\mu_{r1} \pm \sigma_{r1}$	$\mu_{q1} \pm \sigma_{q1}$	$\mu_{m1} \pm \sigma_{m1}$
1MM	$\mu_{r2} \pm \sigma_{r2}$	$\mu_{q2} \pm \sigma_{q2}$	$\mu_{m2} \pm \sigma_{m2}$
10MM	$\mu_{r3} \pm \sigma_{r3}$	$\mu_{q3} \pm \sigma_{q3}$	$\mu_{m3} \pm \sigma_{m3}$

Cuadro 1: Ejemplo de estructura para la tablas de resultados.

3. Informe

1. **Abstract o Resumen**: es el resumen del trabajo.
2. **Introducción**: describe el problema que tratas de resolver. (10 %)
3. **Desarrollo**: aquí se describe el diseño e implementación de los programas necesarios para realizar sus experimentos. Además, presenta una descripción y ejemplos de cómo usar tus programas. No es necesario poner pantallazos, hay que describir tu solución o soluciones. Debe haber coherencia entre tu informe y el código implementado. (40 %)
4. **Resultados Experimentales y Discusión**: aquí se presentan los resultados, pero lo más importante es analizarlos. Se debe observar y describir el comportamiento de los métodos. Comenta si el uso de *RadixSort* es efectivo o no.

   Por favor, en esta sección deben incluir las características de la computadora sobre la que se realizaron los experimentos. (40 %).
5. **Conclusiones**: ideas o hallazgos principales sobre el trabajo. (10 %)

4. Restricciones

1. Pueden trabajar en grupos de 2 estudiantes.
2. Todos los programas deben ser propios, permitiendo solamente utilizar el código disponible en el repositorio del curso `https://github.com/jmsaavedrar/eda_cpp/`.
3. El hallazgo de plagio será penalizado con nota 1.0, para todos los grupos involucrados.
4. Todos las implementaciones deben ser realizadas en C++, excepto los código de visualización de datos.
5. **La entrega del informe es obligatorio**. Un trabajo sin informe no será calificado, asignando la nota mínima igual a 1.0.

5. Entrega

La entrega se debe realizar por canvas hasta el viernes 30 de septiembre, 2025, 23:50 hrs. La entrega debe incluir:

1. Código fuente (en C++), junto a un README con los pasos de compilación.
2. Informe