Universidad de Guadalajara

Centro Universitario de Ciencias Exactas e Ingenierías

IL365 - Estructura de Datos - D01

Actividad de Aprendizaje #07

Métodos de Ordenamiento Recursivos

Alumna: Cervantes Araujo Maria Dolores

Código: 217782452

Fecha de Elaboración: 10 marzo de 2023

Autoevaluación			
Concepto	**Si**	**No**	**Acumulación**
Bajé el trabajo de internet o alguien me lo pasó (aunque sea de forma parcial)	**-100 pts**	**0 pts**	**0**
Incluí el código fuente **en formato de texto (sólo si funciona cumpliendo todos los requerimientos)**	**+25pts**	**0 pts**	**25**
Incluí las **impresiones de pantalla (sólo si funciona cumpliendo todos los requerimientos)**	**+25pts**	**0 pts**	**25**
Incluí una **portada** que identifica mi trabajo (nombre, código, materia, fecha, título)	**+25 pts**	**0 pts**	**25**
Incluí una **descripción y conclusiones** de mi trabajo	**+25 pts**	**0 pts**	**25**
		Suma:	**100**

Introducción:

La actividad de esta semana conllevo la reafirmación del uso de los métodos de ordenamiento, pero se agregó la recursividad de la mano de dos métodos más (MergeSort y QuickSort), los cuales son más rápidos al momento de ejecutarse; ya que utilizan una técnica coloquialmente llamada “Divide y Vencerás” haciendo intercalaciones entre ambos extremos de la lista, esto nos permite evaluar más valores y ordenarlos en un menor tiempo, el Quicksort separa parámetros y hace ubicación del pivote para cada sección que ordena.

Además del uso de todos los métodos de ordenamiento y la recursividad, a motivo de comparación creamos una función para medir el tiempo en que cada método demora en terminar de ordenar una lista lo suficientemente grande, esto a fin de evaluar cuál método nos conviene más por eficiencia, aunque también depende de la cantidad de datos con los que se trabajen y el tiempo que tengamos a disposición.

Código Fuente:

Lista.hpp

```
#ifndef LISTA_HPP_INCLUDED
#define LISTA_HPP_INCLUDED
#include <iostream>
#include <string>

template<class L, int ARRAY=100000>
class Lista {

    private:
        L data[ARRAY];
        int contLast;
        bool validPosition(const int&);
        void copyAll(const Lista<L, ARRAY>&);

        void swapData(L&, L&);
        void sortDataMerge(const int&, const int&);
        void sortDataQuick(const int&, const int&);

    public:
        Lista();
        Lista(const Lista<L, ARRAY>&);

        bool listVacia();
        bool listFull();
        bool isSorted() const;

        void insertList(const int&, const L&);
        void deletelist(const int&);

        ///Posiciones de la lista: Primera, Ãšltima, Anterior, Siguiente y Nula
        int getFirstPosition();
        int getLastPosition();
        int getBeforePosition(const int&);
        int getNextPosition(const int&);

        ///Metodos de busqueda lineal y binario
        int findDatLin(const L&);
        int findDatBin(const L&);

        L retrieve(const int&);

        friend std::iostream& operator >> (std::iostream&, const L&);
        friend std::ostream& operator << (std::ostream&, const L&);
        std::string toString();

        ///Metodos de ordenamiento
        void bubbleSort();
        void insrtSort();
```

```
        void selectSort();
        void shellSort();
        void mergeSort();
        void quickSort();

        Lista<L, ARRAY>& operator = (const Lista<L, ARRAY>&);
    };
#endif // LISTAA_HPP_INCLUDED

///IMPLEMENTACIÃ"N
using namespace std;

template<class L,int ARRAY>
void Lista<L, ARRAY>::copyAll(const Lista<L,ARRAY>& obj) {
    int i=0;
    while(i <= obj.contLast) {
        data[i] = obj.data[i];

        i++;
        }
    contLast = obj.contLast;
    }

template<class L, int ARRAY>
void Lista<L,ARRAY>::swapData(L& a, L& b) {
    L aux(a);
    a = b;
    b = aux;
    }

template<class L,int ARRAY>
Lista<L, ARRAY>::Lista() : contLast(-1) { }

template<class L,int ARRAY>
Lista<L, ARRAY>::Lista(const Lista<L,ARRAY>& obj) { }

template<class L, int ARRAY>
bool Lista<L,ARRAY>::listVacia() {
    return contLast == -1;
    }

template<class L, int ARRAY>
bool Lista<L,ARRAY>::listFull() {
    return contLast == 999999;
    }

template<class L, int ARRAY>
bool Lista<L,ARRAY>::isSorted() const {
    int i(0), j(contLast);
    bool flag;
    while(i < j) {
        if(data[i]<=data[i+1]) {
            flag = true;
```

```
            }
        else {
            return false;
            }
        i++;
        }
    return true;
    }

template<class L, int ARRAY>
bool Lista<L, ARRAY>::validPosition(const int& position) {
    return position >= 0 && position <= contLast;
    }

///METODO INSERTAR
template<class L, int ARRAY>
void Lista<L,ARRAY>::insertList(const int& position, const L& obj) {
    int x=contLast;
    try {
        if(listFull()) {
            throw "Desbordamiento de Datos";
            }
        else if(position != -1 && !validPosition(position)) {
            throw ("Posicion Invalida");
            return;
            }
        else {

            while(x>position) {
                data[x+1]=data[x];
                x--;
                }
            data[position+1]= obj;
            }
        contLast++;
        }
    catch(const char *error) {
        cout<<error<<endl;
        getchar();
        }
    }

///METODO ELIMINAR
template<class L, int ARRAY>
void Lista<L, ARRAY>::deletelist(const int& position) {
    int x(position);
    try {
        if(listVacia()) {
            throw "Insuficiencia de Datos";
            }
        else if(!validPosition(position)) {
            throw "Posicion Invalida";
            }
```

```
        else {
            while(x<contLast) {
                data[x]=data[x+1];
                x++;
                }
            contLast--;
            }
        }
    catch(const char *error) {
        cout<<error<<endl;
        getchar();
        }
    }

///RECUPERAR
template<class L, int ARRAY>
L Lista<L, ARRAY>::retrieve(const int& position) {
    try {
        if(!validPosition(position)) {
            throw "Posicion invalida en la lista";
            }
        }
    catch(const char *error) {
        cout<<error<<endl;
        }
    return data[position];
    }

///PRIMERA POSICIÃ"N
template<class L, int ARRAY>
int Lista<L, ARRAY>::getFirstPosition() {
    if(listVacia()) {
        return -1;
        }
    return -1;
    }

///ÃšLTIMA POSICIÃ"N
template<class L, int ARRAY>
int Lista<L, ARRAY>::getLastPosition() {
    return contLast;
    }

///ANTES DE CIERTA POSICIÃ"N
template<class L, int ARRAY>
int Lista<L, ARRAY>::getBeforePosition(const int& position) {
    if(position == getFirstPosition() && !validPosition(position)) {
        return -1;
        }
    return position-1;
    }

///DESPUÃ‰S DE CIERTA POSICIÃ"N
```

```
template<class L, int ARRAY>
int Lista<L,ARRAY>::getNextPosition(const int& position) {
    if(position == getLastPosition() && !validPosition(position)) {
        return -1;
        }
    return position;
    }

template<class L,int ARRAY>
void Lista<L, ARRAY>::bubbleSort() {
    int i(contLast), j;
    bool flag;
    do {
        j=0;
        flag = false;
        while(j<i) {
            if(data[j]>data[j+1]) {
                swapData(data[j], data[j+1]);
                flag= true;
                }
            j++;
            }
        i--;
        }
    while(flag);
    }

template<class L,int ARRAY>
void Lista<L, ARRAY>::insrtSort() {
    int i(1), j;
    L aux;

    while(i<= contLast) {
        aux = data[i];
        j=i;

        while(j>0 && aux < data[j-1]) {
            data[j] = data[j-1];
            j--;
            }
        if(i!=j) {
            data[j] = aux;
            }
        i++;
        }
    }

template<class L,int ARRAY>
void Lista<L, ARRAY>::selectSort() {
    int i(0), j, m;

    while(i<contLast) {
        m = i;
```

```
        j = i + 1;
        while(j<=contLast) {
            if(data[j] < data[m]) {
                m = j;
                }
            j++;
            }
        if(i!=m) {
            swapData(data[i], data[m]);
            }
        i++;
        }
    }

template<class L,int ARRAY>
void Lista<L, ARRAY>::shellSort() {
    float factor(1.0 / 2.0);
    int dif((contLast+1)*factor), i, j;

    while(dif>0) {
        i = dif;

        while(i<=contLast) {
            j=i;

            while(j>= dif && data[j-dif] > data[j]) {
                swapData(data[j-dif], data[j]);
                j-=dif;
                }
            i++;
            }
        dif*=factor;
        }
    }

template<class L,int ARRAY>
void Lista<L, ARRAY>::mergeSort() {
    sortDataMerge(0, contLast);
    }

template<class L, int ARRAY>
void Lista<L, ARRAY>::sortDataMerge(const int& leftEdg, const int& rightEdg) {
    ///Criterio de paso
    if(leftEdg>=rightEdg) {
        return;
        }

    ///Divide y venceras. Llamados recursivos
    int m((leftEdg+rightEdg)/2);
    sortDataMerge(leftEdg, m);
    sortDataMerge(m+1, rightEdg);

    ///Copy aux
```

```
    static L aux[ARRAY];
    int n(leftEdg);

    while(n<=rightEdg) {
        aux[n] = data[n];
        n++;
        }

    ///Intercalacion
    int i(leftEdg), j(m+1), x(leftEdg);

    while(i <= m && j <= rightEdg) {
        while(i <= m && aux[i] <= aux[j]) {
            data[x++] = aux[i++];
            }
        if(i<=m) {
            while(j <= rightEdg && aux[j] <= aux[i]) {
                data[x++] = aux[j++];
                }
            }
        }

    while(i<=m) {
        data[x++] = aux[i++];
        }

    while(j <= rightEdg) {
        data[x++] = aux[j++];
        }
    }

template<class L,int ARRAY>
void Lista<L, ARRAY>::quickSort() {
    sortDataQuick(0, contLast);
    }

template<class L, int ARRAY>
void Lista<L, ARRAY>::sortDataQuick(const int& leftExt, const int& rightExt) {
    if(leftExt >= rightExt) {
        return;
        }

    ///SeparaciÃ³n de parametros y ubicaciÃ³n del pivote
    int i(leftExt), j(rightExt);

    while(i < j) {
        while(i < j && data[i] <= data[rightExt]) {
            i++;
            }
        while(i < j && data[j] >= data[rightExt]) {
            j--;
            }
        if(i!=j) {
```

```
            swapData(data[i], data[j]);
            }
        }

    if(i != rightExt) {
        swapData(data[i], data[rightExt]);
        }

    sortDataQuick(leftExt, i-1);
    sortDataQuick(i+1, rightExt);

    }

template<class L, int ARRAY>
std::iostream& operator >> (std::iostream& is, const L& obj) {
    is >> obj.data;
    return is;
    }

template<class L, int ARRAY>
std::ostream& operator << (std::ostream& os, const L& obj) {
    os << obj.data;
    return os;
    }

template<class L, int ARRAY>
Lista<L, ARRAY>& Lista<L,ARRAY>::operator = (const Lista<L, ARRAY>& obj) {
    copyAll(obj);
    return *this;
    }

template<class L,int ARRAY>
string Lista<L, ARRAY>::toString() {
    string listComplete;
    int i=0;
    while(i<=contLast) {
        listComplete += data[i];
        listComplete += '\n';
        i++;
        }
    return listComplete;
    }
```

Integer.hpp

```
#ifndef INTEGER_HPP_INCLUDED
#define INTEGER_HPP_INCLUDED
#include <iostream>

class Integer {
    private:
        unsigned int data;

    public:
        Integer();
        Integer(int&);
        Integer(const Integer&);

        void setData(const int& );
        int getData() const;

        std::string toString() const;

        //OPERATORS
        Integer& operator = (const Integer&);
        Integer& operator = (const int&);

        Integer& operator *= (const Integer&);
        Integer& operator /= (const Integer&);
        Integer& operator += (const Integer&);
        Integer& operator -= (const Integer&);
        Integer& operator %= (const Integer&);

        Integer& operator - (const Integer&);
        Integer& operator + (const Integer&);
        Integer& operator * (const Integer&);
        Integer& operator / (const Integer&);
        Integer& operator % (const Integer&);

        Integer& operator ++();
        Integer operator ++(int);
        Integer& operator --();
        Integer operator --(int);

        bool operator == (const Integer&) const;
        bool operator != (const Integer&) const;
        bool operator >= (const Integer&) const;
        bool operator <= (const Integer&) const;
        bool operator > (const Integer&) const;
        bool operator < (const Integer&) const;

        int compareTo(const Integer&) const;
        static int compare(const Integer&, const Integer&);
        friend std::istream& operator >> (std::istream&, Integer&);
        friend std::ostream& operator << (std::ostream&, Integer&);
```

```
    };

#endif // INTEGER_HPP_INCLUDED
```

Integer.cpp

```
#include "Integer.hpp"

using namespace std;

Integer::Integer() {
    setData(0);
    }

Integer::Integer(int& obj) {
    data = obj;
    }

Integer::Integer(const Integer& obj) {
    data = obj.data;
    }

void Integer::setData(const int& obj) {
    data = obj;
    }

int Integer::getData() const {
    return data;
    }

std::string Integer::toString() const {
    string result;
    result += to_string(data);
    return result;
    }

Integer& Integer::operator = (const Integer& obj) {
    data = obj.data;
    return *this;
    }

Integer& Integer::operator = (const int& obj) {
    data = obj;
    return *this;
    }

Integer& Integer::operator -= (const Integer& obj) {
    data = data - obj.data;
    return *this;
    }

Integer& Integer::operator %= (const Integer& obj) {
```

```
    data = data % obj.data;
    return *this;
    }

Integer& Integer::operator - (const Integer& obj) {
    data += obj.data;
    return *this;
    }

Integer& Integer::operator / (const Integer& obj) {
    data /= obj.data;
    return *this;
    }

Integer& Integer::operator % (const Integer& obj) {
    data %= obj.data;
    return *this;
    }

bool Integer::operator == (const Integer& obj) const {
    return data == obj.data;
    }

bool Integer::operator != (const Integer& obj) const {
    return data != obj.data;
    }

bool Integer::operator < (const Integer& obj) const {
    return data < obj.data;
    }

bool Integer::operator <= (const Integer& obj) const {
    return data <= obj.data;
    }

bool Integer::operator > (const Integer& obj) const {
    return data > obj.data;
    }

bool Integer::operator >= (const Integer& obj) const {
    return data >= obj.data;
    }

Integer& Integer::operator * (const Integer& obj) {
    data *= obj.data;
    return *this;
    }

Integer& Integer::operator + (const Integer& obj) {
    data += obj.data;
    return *this;
    }
```

```
Integer& Integer::operator /= (const Integer& obj) {
    data /= obj.data;
    return *this;
    }

Integer& Integer::operator += (const Integer& obj) {
    data += obj.data;
    return *this;
    }

Integer& Integer::operator *= (const Integer& obj) {
    data *= obj.data;
    return *this;
    }

Integer& Integer::operator++() {
    data++;
    return *this;
    }

Integer Integer::operator++(int) {
    Integer tmp(*this);
    operator++();
    return tmp;
    }

Integer& Integer::operator--() {
    data--;
    return *this;
    }

Integer Integer::operator--(int) {
    Integer tmp(*this);
    operator--();
    return tmp;
    }

std::istream& operator >> (std::istream& is, Integer& obj) {
    is >> obj.data;
    return is;
    }

std::ostream& operator << (std::ostream& os, Integer& obj) {
    os << obj.data;
    return os;
    }
```

Menu.hpp

```
#ifndef MENU_HPP_INCLUDED
#define MENU_HPP_INCLUDED
#include <string>
#include <random>
#include <chrono>
#include <iostream>
#include <windows.h>
#include <functional>
#include "Lista.hpp"
#include "Integer.hpp"
#define MAX 100000

class Menu {
    private:
        void view(Lista<Integer>&);
    public:
        Menu(Lista<Integer>&);

    };
#endif // MENU_HPP_INCLUDED
```

Menu.cpp

```
#include "Menu.hpp"

using namespace std;

Menu::Menu(Lista<Integer> &firstList) {
    system("Color E5");
    view(firstList);
    }

void Menu::view(Lista<Integer> &firstList) {

    std::chrono::steady_clock::time_point begin, end;
    Lista<Integer> secondList;
    Integer ListNumbers;
    default_random_engine
generator(chrono::system_clock::now().time_since_epoch().count());
    uniform_int_distribution<int>distribution(0,1000000);
    auto random = bind(distribution, generator);

    cout<<"LLENANDO LISTA..."<<endl;
    int i=0;
    while(i<MAX) {
        ListNumbers = random(generator);
        firstList.insertList(firstList.getLastPosition(), ListNumbers);
        i++;
        }
```

```
    cout<<"\t\t...LISTA LLENA"<<endl;

    ///BUBBLE
    cout<<"\n---------------- BUBBLESORT ----------------"<<endl;
    secondList = firstList;
    if(secondList.isSorted()) {
        cout<<"Lista Ordenada"<<endl;
        }
    else {
        cout<<"Lista No Ordenada :("<<endl;
        }
    begin = chrono::steady_clock::now();
    secondList.bubbleSort();
    end = chrono::steady_clock::now();
    if(secondList.isSorted()) {
        cout<<"Estado Actual: ORDENADA"<<endl;
        cout<<"Tiempo de ejecucion en milisegundos: "<<
chrono::duration_cast<chrono::milliseconds>(end - begin).count()<<" ms"<<endl;
        }


    ///INSERT
    cout<<"\n---------------- INSERTSORT ----------------"<<endl;
    secondList = firstList;
    if(secondList.isSorted()) {
        cout<<"Lista Ordenada"<<endl;
        }
    else {
        cout<<"Lista No Ordenada :("<<endl;
        }

    begin = chrono::steady_clock::now();
    secondList.insrtSort();
    end = chrono::steady_clock::now();
    if(secondList.isSorted()) {
        cout<<"Estado Actual: ORDENADA"<<endl;
        cout<<"Tiempo de ejecucion en milisegundos: "<<
chrono::duration_cast<chrono::milliseconds>(end - begin).count()<<" ms"<<endl;
        }


    ///SELECT
    cout<<"\n---------------- SELECTSORT ----------------"<<endl;
    secondList = firstList;
    if(secondList.isSorted()) {
        cout<<"Lista Ordenada"<<endl;
        }
    else {
        cout<<"Lista No Ordenada :("<<endl;
        }
    begin = chrono::steady_clock::now();
    secondList.selectSort();
```

```
    end = chrono::steady_clock::now();
    if(secondList.isSorted()) {
        cout<<"Estado Actual: ORDENADA"<<endl;
        cout<<"Tiempo de ejecucion en milisegundos: "<<
chrono::duration_cast<chrono::milliseconds>(end - begin).count()<<" ms"<<endl;
        }

    ///SHELL
    cout<<"\n---------------- SHELLSORT ----------------"<<endl;
    secondList = firstList;
    if(secondList.isSorted()) {
        cout<<"Lista Ordenada"<<endl;
        }
    else {
        cout<<"Lista No Ordenada :("<<endl;
        }
    begin = chrono::steady_clock::now();
    secondList.shellSort();
    end = chrono::steady_clock::now();
    if(secondList.isSorted()) {
        cout<<"Estado Actual: ORDENADA"<<endl;
        cout<<"Tiempo de ejecucion en milisegundos: "<<
chrono::duration_cast<chrono::milliseconds>(end - begin).count()<<" ms"<<endl;
        }

    ///MERGE
    cout<<"\n---------------- MERGESORT ----------------"<<endl;
    secondList = firstList;
    if(secondList.isSorted()) {
        cout<<"Lista Ordenada"<<endl;
        }
    else {
        cout<<"Lista No Ordenada :("<<endl;
        }
    begin = chrono::steady_clock::now();
    secondList.mergeSort();
    end = chrono::steady_clock::now();
    if(secondList.isSorted()) {
        cout<<"Estado Actual: ORDENADA"<<endl;
        cout<<"Tiempo de ejecucion en milisegundos: "<<
chrono::duration_cast<chrono::milliseconds>(end - begin).count()<<" ms"<<endl;
        }

    ///QUICK
    cout<<"\n---------------- QUICKSORT ----------------"<<endl;
    secondList = firstList;
    if(secondList.isSorted()) {
        cout<<"Lista Ordenada"<<endl;
        }
    else {
        cout<<"Lista No Ordenada :("<<endl;
        }
    begin = chrono::steady_clock::now();
```

```
    secondList.quickSort();
    end = chrono::steady_clock::now();
    if(secondList.isSorted()) {
        cout<<"Estado Actual: ORDENADA"<<endl;
        cout<<"Tiempo de ejecucion en milisegundos: "<<
chrono::duration_cast<chrono::milliseconds>(end - begin).count()<<" ms"<<endl;
        }

    }
```

Main.cpp

```
#include "Menu.hpp"
int main() {
    Lista<Integer> m;
    Menu start(m);
    }
```

Impresiones de Pantalla:

```
"C:\Users\cerva\Escritorio\F.Prog\Estructura de datos\Actividad7\bin\Debug\Actividad7.exe"
LLENANDO LISTA...
               ...LISTA LLENA

--------------- BUBBLESORT ---------------
Lista No Ordenada :(
Estado Actual: ORDENADA
Tiempo de ejecucion en milisegundos: 43009 ms

--------------- INSERTSORT ---------------
Lista No Ordenada :(
Estado Actual: ORDENADA
Tiempo de ejecucion en milisegundos: 9604 ms

--------------- SELECTSORT ---------------
Lista No Ordenada :(
Estado Actual: ORDENADA
Tiempo de ejecucion en milisegundos: 12517 ms

--------------- SHELLSORT ---------------
Lista No Ordenada :(
Estado Actual: ORDENADA
Tiempo de ejecucion en milisegundos: 40 ms

--------------- MERGESORT ---------------
Lista No Ordenada :(
Estado Actual: ORDENADA
Tiempo de ejecucion en milisegundos: 22 ms

--------------- QUICKSORT ---------------
Lista No Ordenada :(
Estado Actual: ORDENADA
Tiempo de ejecucion en milisegundos: 16 ms

Process returned 0 (0x0)   execution time : 65.390 s
Press any key to continue.
```

Resumen Personal

Al término de esta actividad puedo concluir que, dado los resultados obtenidos en la ejecución del programa, el método BubbleSort es el que más tarda cuando hablamos de una cantidad grande de datos, por lo que a mí parecer pierde eficiencia; en su lugar, utilizaría el MergeSort o el QuickSort porque son métodos más rápidos y creo que son más convenientes cuando trabajamos con bases de datos enormes.

El tema de la recursividad fue el que me causo un poco de conflicto al momento de realizar la actividad ya que al inicio no me daba el método QuickSort, me llevo de experiencia que hay que tener buen manejo de variables y parámetros cuando trabajamos con este método de lo contrario te congela la ejecución, lo que me fallo fue un error de dedo, ya que la condicional que estaba utilizando para ubicar mi pivote lo había colocado al revés; de ahí en más pudiera decir que fue una práctica sencilla, solo tuve que volver a recordar la estructura para utilizar plantillas (template) e integrar la librería “Chrono” para tomar los tiempos de cada ordenamiento.