LABORATORIO 1

Análisis de Algoritmos - Insertion Sort

Ejercicio 1

Implementar el algoritmo Insertion Sort, según el pseudo código presentando en el libro de Introducción a los Algoritmos de Thomas H. Cormen. [1]

INSERTION − SORT(*A*) :
for *j* = 2 **to** *A.length* **do**
key = *A*[*j*]
// Insert *A*[*j*] into the sorted sequence *A*[1..*j* − 1]

i = *j* − 1

while *i* > 0 **and** *A*[*i*] > *key* **do**
A[*i* + 1] = *A*[*i*] *i* = *i* − 1
end while
A[*i* + 1] = *key*

end for

SOLUCION:

```
#include <iostream>
#include <vector>
using namespace std;

void insertionSort(std::vector<int> &list);
void printVector(vector<int> &lista);

int main()
{
    vector<int> vector1={4,2,65,8,6,3};
    insertionSort(vector1);
    printVector(vector1);
    return 0;
}

void insertionSort(vector<int> &lista)
{
    for (int i = 1; i < lista.size(); i++) {
        int key = lista[i];
        int j = i - 1;
        while (j >= 0 && lista[j] > key) {
```

```
            lista[j + 1] = lista[j];
            j--;
        }
        lista[j + 1] = key;
    }
}
void printVector(vector<int> &lista)
{
    for (int i=0; i<lista.size(); i++) {
        cout << lista[i]<<" ";
    }
}
```

EJECUCION:

```
2 3 4 6 8 65
```

CONCLUSION:

Se implemento el algoritmo de insertion sort, según el seudo código con ciertas modificaciones, y se hizo una función aparte, para verificar el funcionamiento de este.

Ejercicio 2

El bucle *While* del algoritmo Insertion Sort, a veces es visto como una búsqueda secuencial, donde se está tratando de localizar una posición adecuada para un elemento. Modifique este comportamiento de búsqueda secuencial por el algoritmo de búsqueda binaria.

```
#include <iostream>
#include <time.h>
using namespace std;

void imprimir (int lista[],int tam);
void insertionSort(int lista[], int tam);
int binarySearch(int a[], int x, int low, int high);

int main()
{
    int n=5;
    int lista[n];
    srand(time(NULL));
    for(int i=0;i< n;i++)
```

```
        lista[i]=rand()%100;

    insertionSort(lista,n);
    imprimir(lista,n);
    return 0;
}
void insertionSort(int lista[], int tam)
{
    for (int i = 1; i < tam; i++) {
        int key = lista[i];
        int j = i - 1;
        int pos = binarySearch(lista, key, 0, j);
        while (j >= pos)
        {
            lista[j + 1] = lista[j];
            j--;
        }
        lista[j + 1] = key;
    }
}
int binarySearch(int a[], int x, int low, int high)
{
    if (high <= low)
        return (x > a[low]) ? (low + 1) : low;

    int mid = (low + high) / 2;
    if (x == a[mid])
        return mid + 1;

    if (x > a[mid])
        return binarySearch(a, x,mid + 1, high);

    return binarySearch(a, x, low,mid - 1);
}
void imprimir (int lista[],int tam)
{
    for (int i=0;i < tam;i++)
        cout<<lista[i]<<" ";
}
```

Ejercicio 3

En la siguiente actividad se realizar el análisis de la complejidad del algoritmo Insertion Sort, para esto se tendrá que ejecutar repetidamente el algoritmo implementado con diferentes cantidades de datos.

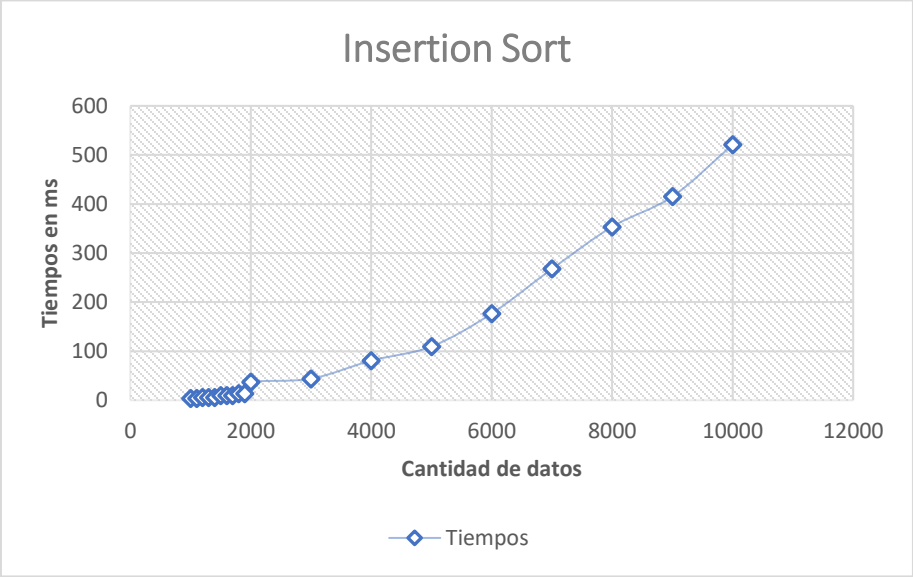
De manera aleatoria, generar una lista de números n = 1000 y procesarlo con el algoritmo de ordenamiento. Registre el tiempo empleado en procesar esa cantidad de datos. Una vez finalizado, aumente la cantidad de datos a trabajar (n = 2000) y repita el proceso de ordenamiento, guardando el tiempo empleado.

```cpp
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <chrono>
#include <fstream>
using namespace std;

void imprimir (int lista[],int tam);
void insertionSort(int lista[], int tam);

int main()
{
    srand(time(NULL));
    for(int i=0;i<10;i++)
    {
        int n=1000+i*100;
        int lista[n];
        for(int i=0;i< n;i++)
        {
            lista[i]=rand()%100;
        }

        ofstream archivo("parametros_3.txt");
        auto start = std::chrono::system_clock::now();
        insertionSort(lista,n);
        auto end = std::chrono::system_clock::now();
        std::chrono::duration <float, std::milli> duration = end - start;
        archivo<<duration.count()<<" ms"<<std::endl;
        cout<<n<<" "<<duration.count()<<" ms"<<std::endl;

        //cout<< "Arreglo Ordenado: \n";
        //imprimir(lista,n);
    }
    return 0;
}
void insertionSort(int lista[], int tam)
```

```
{

    for (int i = 1; i < tam; i++) {
        int key = lista[i];
        int j = i - 1;
        while (j >= 0 && lista[j] > key) {
            lista[j + 1] = lista[j];
            j--;
        }
        lista[j + 1] = key;
    }
}

void imprimir (int lista[],int tam){
    for (int i=0;i < tam;i++)
        cout<<lista[i]<<" ";
}
```

CONCLUSION

Se tomaron datos desde los 1000 y se fue incrementando de 100 en 100 hasta 2 mil y luego se fue incrementando de mil en mil mostrando la siguiente grafica.



Como observamos en la gráfica, vemos que mientras mas se va incrementando el tamaño de los datos este tiende a un comportamiento cuadrático.

Ejercicio 4

El algoritmo de la burbuja (*Bubblesort*) es uno de los algoritmos más simples de implementar pero ineficiente. A continuación, se muestra el pseudo código de este algoritmo:

BUBBLESORT(*A*) :
for *i* = 1 **to** *A.length* − 1 **do**
  for *j* = *A.length* **to** *i* + 1 **do** //*j*
    downto *i* + 1 **if** *A*[*j*] < *A*[*j* −
    1] **then**

      exchange *A*[*j*] with *A*[*j* − 1]

    end if end
  for
end for

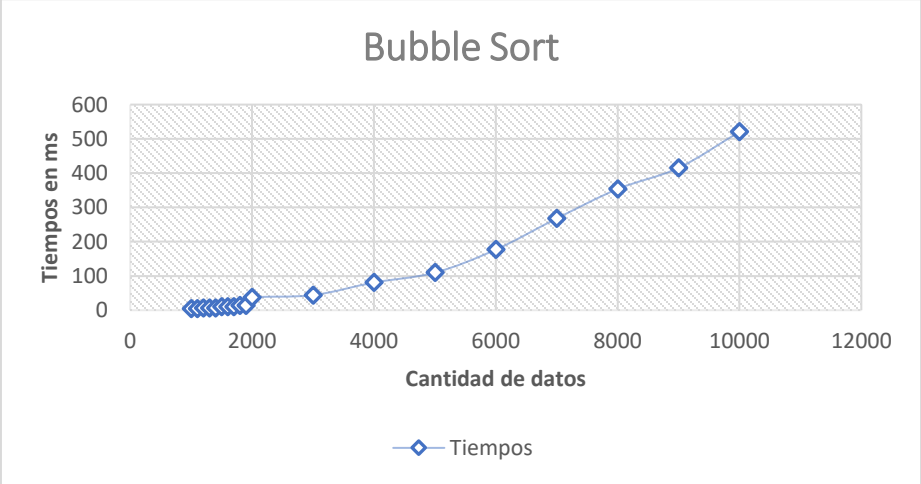
Repetir la actividad del Ejercicio 3 con este algoritmo. Realice un gráfico comparativo entre ambos algoritmos e indique cual podría ser el tiempo de ejecución del algoritmo *Bubblesort*.

```cpp
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <chrono>
#include <fstream>
using namespace std;

void bubbleSort(int list[], int n);
void Imprimir(int list[], int n);

int main()
{

    srand(time(NULL));
    for(int i=0;i<10;i++)
    {
        int n=1000+i*100;
    int lista1[n];
        for(int i=0;i < n; i++)
            lista1[i]=rand()%100;

        ofstream archivo("parametros_4.txt");
        auto start = std::chrono::system_clock::now();
        bubbleSort(lista1, n);
        auto end = std::chrono::system_clock::now();
        std::chrono::duration <float, std::milli> duration = end - start;
        archivo<<duration.count()<<" ms"<<std::endl;
        cout<<n<<" "<<duration.count()<<" ms"<<std::endl;
```

```
        //cout<< "Arreglo Ordenado: \n";
    }
        //Imprimir(lista1, n);
    return 0;
}
void bubbleSort(int list[], int n)
{
    for (int i =0; i <n - 1; i++){
        for (int j =0; j <n - i - 1; j++){
            if (list[j] > list[j + 1])
                swap(list[j], list[j + 1]);
        }
    }
}

void Imprimir(int list[], int n)
{
    for (int i = 0; i <n; i++)
        cout<<list[i] << " ";
    cout<<endl;
}
```

CONCLUSION

Se tomaron datos desde los 1000 y se fue incrementando de 100 en 100 hasta 2 mil y luego se fue incrementando de mil en mil mostrando la siguiente grafica.



Como observamos en la gráfica, del mismo modo que insertion sort, vemos que mientras más se va incrementando el tamaño de los datos este tiende a un comportamiento cuadrático en el peor de los casos, pero para datos pequeños vemos que casi la complejidad es menor.