Universidad Nacional de San Agustín de Arequipa

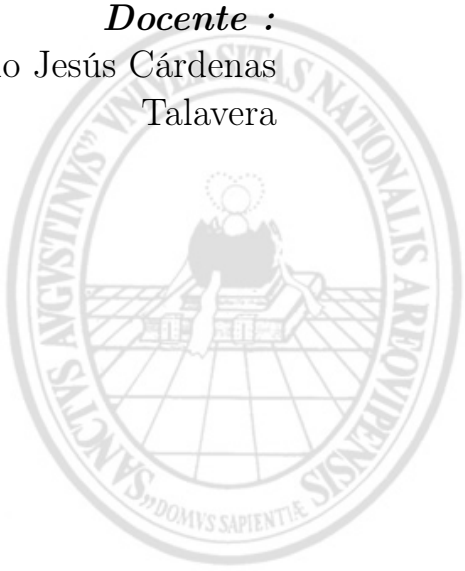
Escuela Profesional de Ciencia de la Computación

Estructura de Datos Avanzados

Laboratorio 1 | Algoritmos y Costo Computacional

Presentado por:
Rushell Vanessa Zavalaga Orozco

Docente :
Rolando Jesús Cárdenas Talavera



1. Actividad

1. Utilizando los archivos adjuntos (DataGen1, DataGen05, DataGen025), utilice los datos para las pruebas de los algoritmos de ordenamiento. Tenga en cuenta la cantidad de datos de cada uno.
2. Implemente los siguientes algoritmos:
   - Bubble sort
   - Heap sort
   - Insertion sort
   - Selection sort
   - Shell sort
   - Merge sort
   - Quick sort
3. Analizar la complejidad computacional de cada uno.
4. Evaluar y comparar sus algoritmos usando los archivos de datos y elabore una(s) gráfica(s) comparativa(s). De utilizar c++, mida el tiempo de ejecución con la función *std::chrono::high resolution clock::now();*

2. Desarrollo

2.1. Bubble Sort

Costo Computacional: $O(n^2)$ Este algoritmo es el de mayor costo, ya que compara elementos adyacentes, hasta n veces. Lo cual daría un resultado de n veces repetir $n-1$ de comparaciones.

```
void bubbleSort(vector<float> &v) {
    for (int i = v.size() - 1; i > 0; i--) {
        bool intercambio = false;
        for (int j = 0; j < i; j++) {
            if (v[j] > v[j + 1]) {
                swap(v[j], v[j + 1]);
                intercambio = true;
            }
        }
        if (!intercambio) break;
    }
}
```

2.2. Heap Sort

Costo Computacional: $O(nlogn)$ Heap Sort consta de dos funciones una la cual construye el heap del array dado lo cual es un costo de $O(n)$ y cada extracción es de $O(logn)$ dando como resultado $O(nlogn)$

```
void make_heap(vector<float> &v, int n, int i){
    int max = i;
    int left = 2*i +1;
    int right = 2*i + 2;

    if(left < n && v[left] > v[max]) max = left;
    if(right< n && v[right] > v[max]) max = right;

    if(max!=i){
        swap(v[i], v[max]);
        make_heap(v, n, max);
    }
}
void heapSort(vector<float> &v){ // O(nlogn)
    int n = v.size();
    for(int i=n/2-1; i>=0; i--) {
        make_heap(v, n, i);
    }

    for(int i=n-1; i>=0; i--){
        swap(v[0],v[i]);
        make_heap(v,i,0);
    }
}
```

2.3. Insertion Sort

Costo Computacional: $O(n^2)$ Ordena cada elemento en su posicion correcto comparando todos los elementos previos. Quiere decir que en el peor caso tiene un costo de $O(n^2)$

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

```
            }
            v[i+1] = key;
        }
    }
```

2.4. Selection Sort

Costo Computacional: $O(nlogn)$ Selecciona el elementos mas pequeño de todo un array y lo intercambia con la primera posicion del array desordenado.

```
void selectionSort(vector<float> &v){
    int n = v.size();
    for(int i=0; i<n-1; i++){
        int minI = i;
        for(int j=i+1; j<n; j++){
            if(v[j]<v[minI]) minI = j;
        }
        if(minI!=i) swap(v[minI], v[i]);
    }
}
```

2.5. Shell Sort

Costo Computacional: $O(nlogn)$ La complejidad de Shell Sort depende de la secuencia del Gap que se elija en este caso la secuencia basica Gap es $n/2$, $n/4$, etc. El peor caso para esto es $O(n^2)$, pero en el mejor de los casos es $O(nlogn)$

```
void shellSort(vector<float> &v){
    int n = v.size();
    for(int gap=n/2; gap>0; gap/=2){
        for(int i=gap; i<n; i++){
            int temp = v.size();
            int j;
            for( j = i; j>=gap && v[j-gap]>temp; j-=gap){
                v[j] = v[j-gap];
            }
            v[j] = temp;
        }
    }
}
```

2.6. Merge Sort

Costo Computacional: $O(nlogn)$ Merge Sort divide el array de forma recursiva hasta llegar a un elemento luego los combina. El proceso de dividir es $O(n)$ y la división ocurrira $O(logn)$ (por que se divide a la mitad recursivamente)

```
void merge(vector<float> &v, int p, int q, int r){
    int n1 = q-p+1;
    int n2 = r-q;
    vector<float> L, R;

    for(int i=0; i<n1; i++){
        L.push_back(v[p+i]);
    }

    for(int i=0; i<n2; i++){
        R.push_back(v[q+i+1]);
    }

    L.push_back(INT_MAX);
    R.push_back(INT_MAX);

    int i=0, j = 0;
    for(int k = p; k<=r; k++){
        if(L[i]<=R[j]){
            v[k] = L[i];
            i++;
        }else{
            v[k] = R[j];
            j++;
        }
    }
}
void mergeSort(vector<float> &v){
    int bajo = 0, alto = v.size()-1;
    for(int m = 1; m<=alto-bajo; m = 2*m){
        for(int n = bajo; n<alto; n+=2*m){
            int from = n;
            int mid = n + m - 1;
            int to = min(n + 2*m - 1, alto);

            merge(v, from, mid, to);
        }
    }
}
```

2.7. Quick Sort

Costo Computacional: $O(nlogn)$ El costo computacional de QuickSort tiene que ver con el pivote que elijamos, si es muy bajo o muy alto, el costo se elevara. En un caso promedio su costo es $O(nlogn)$ y en el peor caso, donde elijamos un mal pivote su complejidad es $O(n^2)$

```
int partition(vector<float> &v, int p, int r){
    float x = v[r];
    int i = p-1;
    for(int j=p; j<=r-1; j++){
        if(v[j]<=x){
            i++;
            swap(v[i], v[j]);
        }
    }
    swap(v[i+1], v[r]);
    return i+1;
}
void quickSort(vector<float> &v, int p, int r){
    if(p<r){
        int q = partition(v, p,r);
        quickSort(v, p, q-1);
        quickSort(v, q+1, r);
    }
}
```

3. Resultados

```
.\DataGen025
 Heap Sort -> 258.303 milisegundos SORTED
 Shell Sort -> 50.1782 milisegundos SORTED
 Merge Sort -> 299.198 milisegundos SORTED
 Quick Sort -> 116.858 milisegundos SORTED
.\DataGen05.txt
 Heap Sort -> 630.009 milisegundos SORTED
 Shell Sort -> 121.282 milisegundos SORTED
 Merge Sort -> 640.493 miclisegundos SORTED
 Quick Sort -> 262.618 milisegundos SORTED
.\DataGen1.txt
 Heap Sort -> 1290.42 milisegundos SORTED
 Shell Sort -> 293.955 milisegundos SORTED
 Merge Sort -> 1502.91 milisegundos SORTED
 Quick Sort -> 509.429 milisegundos SORTED
```

```
.\DataGen025
 Bubble Sort -> 8.64737 segundos SORTED
 Insertion Sort -> 2.6389 segundos SORTED
 Selection Sort -> 3.09096 segundos SORTED
.\DataGen05
 Bubble Sort -> 35.1526 segundos SORTED
 Insertion Sort -> 7.82484 segundos SORTED
 Selection Sort -> 10.838 segundos SORTED
.\DataGen1
 Bubble Sort -> 128.381 segundos SORTED
 Insertion Sort -> 32.8193 segundos SORTED
 Selection Sort -> 42.6339 segundos SORTED
```

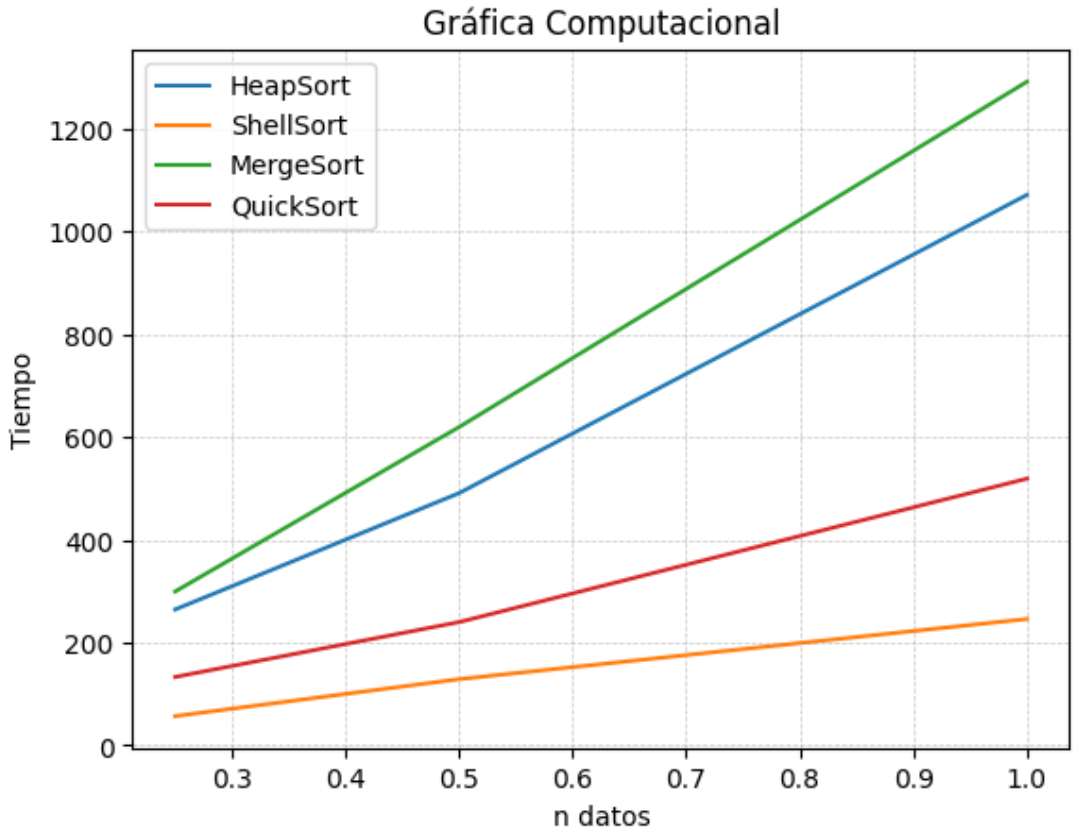


Figura 1: Algoritmos más rapidos

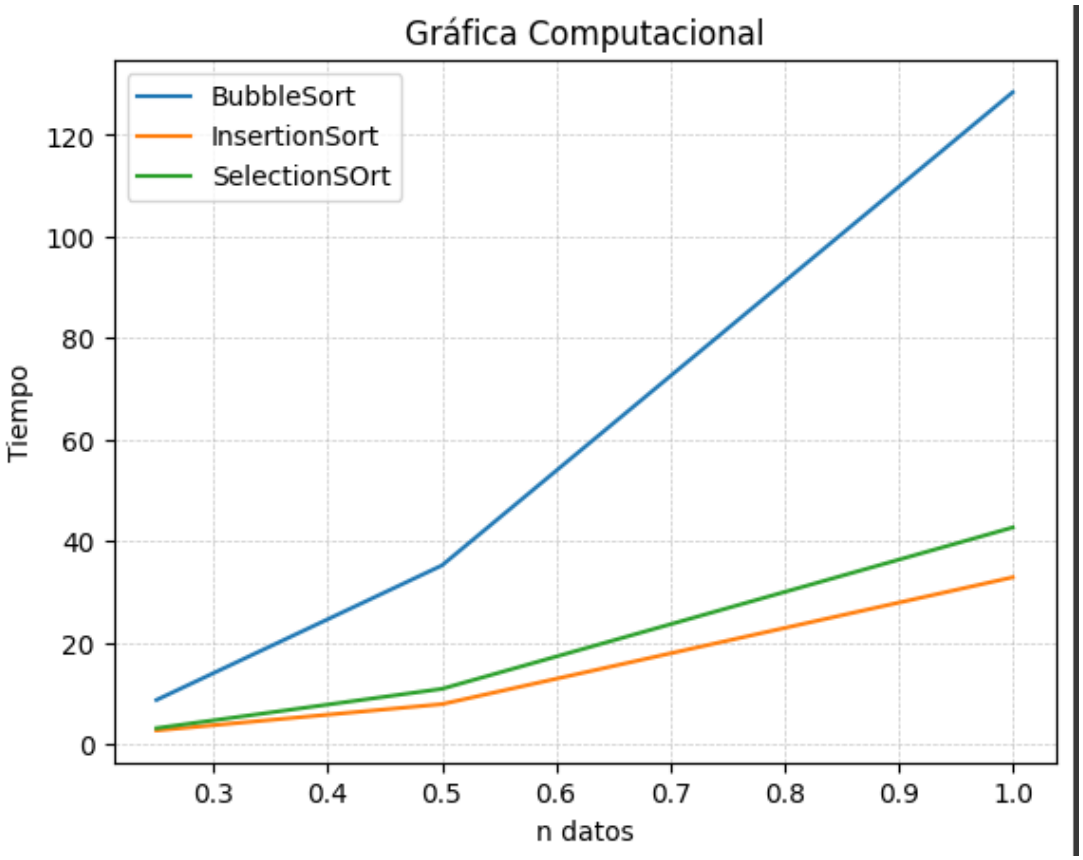


Figura 2: Algoritmos menos rapidos