Instituto Politécnico Nacional - Escuela Superior de Cómputo

Análisis de Algoritmos

Práctica 2: Pruebas a Posteriori (Algoritmos de Búsqueda)

Equipo:
CompilandoConocimiento.com
Integrantes:
Morales López Laura Andrea
Ontiveros Salazar Alan Enrique
Rosas Hernández Óscar Andrés

Profesor:
Franco Martínez Edgardo Adrián

Abril 2018

Índice general

0.1. Introducción 3
0.1.1. Métodos 3
0.1.2. Definición 12
0.1.3. Algoritmo de ordenamiento 12
0.1.4. Algoritmos famosos 12
0.2. Planteamiento del problema 12
0.3. Diseño de la solución 14
0.3.1. Burbuja 14
0.3.2. Inserción 15
0.3.3. Selección 16
0.3.4. Shell 16
0.3.5. Árbol binario de búsqueda 17
0.4. Implementación de la solución 18
0.4.1. Burbuja 18
0.4.2. Inserción 19
0.4.3. Selección 19
0.4.4. Shell 20
0.4.5. Árbol binario de búsqueda 20
0.5. Actividades y pruebas 23
0.5.1. Comparativas individuales 23
0.5.2. Comparativas globales 36
0.5.3. Preguntas 42
0.6. Errores detectados 44

0.7. Posibles mejoras 44
0.8. Conclusiones 45
0.8.1. Alan 45
0.8.2. Óscar 45
0.8.3. Laura 47

Appendices **49**
.1. Estructura de directorios 50
.2. Código fuente original 50
.2.1. `AuxFunctions.c` 50
.2.2. `TreeAuxFunction.c` 51
.2.3. `Time.h` 54
.2.4. `Time.c` 55
.2.5. `BubbleSort.c` 57
.2.6. `InsertionSort.c` 59
.2.7. `SelectionSort.c` 60
.2.8. `ShellSort.c` 61
.2.9. `SortWithBST.c` 62
.2.10. `TestSortAlgorithms.c` 63
.2.11. `Make.py` 66
.3. Compliación y ejecución 70

Bibliografía **71**

0.1. Introducción

Uno de los típicos problemas dentro de un curso de programación es el ordenamiento. Estos algoritmos son la base de muchos otros, además de que tenemos con ellos unas ideas interesantes a usar en otro tipo de algoritmos, como divide y vencerás, las estructuras de datos y los algoritmos aleatorios.

Tenemos que tener en cuenta que las computadoras pasan más tiempo ordenando que haciendo otra cosa, sigue siendo el problema de algoritmo combinatorio, también llamado de optimización combinatoria, más presente en el mundo. Como resultado de su estudio existen varias maneras de realizarlo, cada una con una ventaja sobre las demás.

0.1.1. Métodos

Veamos unos cuantos con un ejemplo de un arco iris.

Como debemos saber el arco iris se ordena por su frecuencia:

Empecemos con un arco iris desordenado que siempre será el mismo para cada método.

Bubble Sort

Este es hacer simplemente un intercambio comparando de dos en dos

Bubble Sort v2

Lo mismo que el anterior solo que asumimos que despues de la primera vuelta el ultimo ya esta ordenado asi que no necesitamos compararlo.

Bubble Sort v3

Simplemente checamos que no este ordenado ya.

Selection Sort

Selecciona el más pequeño y lo coloca hasta el principio.

Insertion Sort

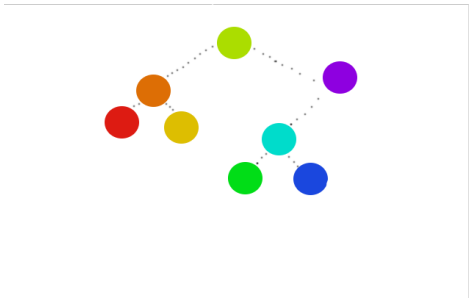
Este es parecido al selection, solo que esta vez buscamos el lugar correcto para cada color en vez de buscar el más pequeño.

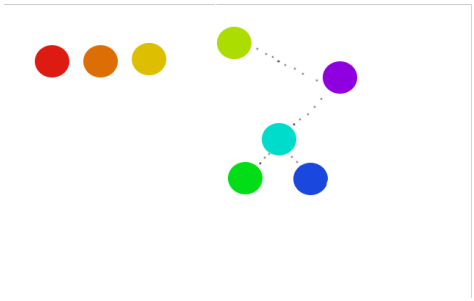
Shell Sort

Un poco mas complejo de entender pero,tomamos saltos compararemos por ejemplo el 1 y 4 y los ordenaremos, despues el 4 y el 8 y asi sucesivamente hasta que todos esten ordenados, bajamos el salto a 3 y asi sucesivamente.

BTS

Metemos el arreglo en un arbol y lo recorremos en inorden.





0.1.2. Definición

Dado un arreglo $A[0, 1, \ldots, n-1]$ de longitud $n \in \mathbb{N}$ de cualquier tipo de elementos sobre los cuales es posible aplicar una relación de orden total $\leq$, decimos que está **ordenado** si [1]:

$$i < j \quad \Longrightarrow \quad A[i] \leq A[j]$$

Es decir, si A está ordenado podemos decir que $A[0] \leq A[1] \leq \cdots \leq A[n-1]$. Por convención, consideraremos a los arreglos de tamaño 0 y 1 como ordenados trivialmente.

En esta práctica consideraremos a los elementos de A como números enteros.

0.1.3. Algoritmo de ordenamiento

Es un algoritmo que recibe un arreglo $A[0, 1, \ldots, n-1]$ y devuelve una permutación ordenada de A, llamémosle B, tal que $B[0] \leq B[1] \leq \cdots \leq B[n-1]$. En la práctica el arreglo B se sobreescribe a A. Más formalmente, este algoritmo calcula de forma indirecta una función de *permutación ordenadora* de los índices de A, $\sigma : \{0, 1, \ldots, n-1\} \to \{0, 1, \ldots, n-1\}$, tal que $A[\sigma(0)] \leq A[\sigma(1)] \leq \cdots \leq A[\sigma(n-1)]$.

Decimos que un algoritmo de ordenamiento es **estable** si:

$$i < j \text{ y } A[i] = A[j] \quad \Longrightarrow \quad \sigma(i) \leq \sigma(j)$$

Es decir, los elementos iguales mantienen sus posiciones relativas despúes de ordenar el arreglo.

0.1.4. Algoritmos famosos

0.2. Planteamiento del problema

Con base en el archivo de entrada proporcionado que tiene **10,000,000 de números diferentes**, ordenarlo bajo los siguientes métodos de ordenamiento y comparar experimentalmente las complejidades de estos:

- Burbuja (Bubble Sort)
  - Burbuja Simple
  - Burbuja Optimizada
- Inserción (Insertion Sort)
- Selección (Selection Sort)

- Shell (Shell Sort)
- Ordenamiento con árbol binario de búsqueda (Tree Sort)

0.3. Diseño de la solución

0.3.1. Burbuja

Algorithm 1 Burbuja Simple

```
1: procedure BubbleSortV1(A, n)
2:     for i ← 0 hasta n − 2 do
3:         for j ← 0 hasta n − 2 do
4:             if A[j] > A[j + 1] then
5:                 aux ← A[j]
6:                 A[j] ← A[j + 1]
7:                 A[j + 1] ← aux
8:             end if
9:         end for
10:    end for
11: end procedure
```

Algorithm 2 Burbuja Optimizada

```
1: procedure BubbleSortV2(A, n)
2:     for i ← 0 hasta n − 2 do
3:         for j ← 0 hasta n − i − 2 do
4:             if A[j] > A[j + 1] then
5:                 aux ← A[j]
6:                 A[j] ← A[j + 1]
7:                 A[j + 1] ← aux
8:             end if
9:         end for
10:    end for
11: end procedure
```

Algorithm 3 Burbuja Optimizada

```
1: procedure BubbleSortV3(A, n)
2:     for i ← 0 hasta n − 2 do
3:         cambio ← NO
4:         for j ← 0 hasta n − i − 2 do
5:             if A[j] > A[j + 1] then
6:                 aux ← A[j]
7:                 A[j] ← A[j + 1]
8:                 A[j + 1] ← aux
9:                 cambio ← SÍ
10:            end if
11:        end for
12:        if cambio = NO then
13:            Salir
14:        end if
15:    end for
16: end procedure
```

0.3.2. Inserción

Algorithm 4 Inserción

```
1: procedure InsertionSort(A, n)
2:     for i ← 1 hasta n − 1 do
3:         j ← i
4:         Temp ← A[i]
5:         while j > 0 y Temp < A[j − 1] do
6:             A[j] ← A[j − 1]
7:             j ← j − 1
8:         end while
9:         A[j] ← Temp
10:    end for
11: end procedure
```

0.3.3. Selección

Algorithm 5 Selección

```
1: procedure SelectionSort(A, n)
2:     for i ← 0 hasta n − 2 do
3:         Smallest ← i
4:         for j ← i + 1 hasta n − 1 do
5:             if A[j] < A[Smallest] then
6:                 Smallest ← j
7:             end if
8:         end for
9:         aux ← A[Smallest]
10:        A[Smallest] ← A[i]
11:        A[i] ← aux
12:    end for
13: end procedure
```

0.3.4. Shell

Algorithm 6 Shell

```
1: procedure ShellSort(A, n)
2:     Gap ← ⌊n/2⌋
3:     while Gap > 0 do
4:         for i ← Gap hasta n − 1 do
5:             j ← i
6:             Temp ← A[i]
7:             while j ≥ Gap y Temp < A[j − Gap] do
8:                 A[j] ← A[j − Gap]
9:                 j ← j − Gap
10:            end while
11:            A[j] ← Temp
12:        end for
13:        Gap ← ⌊Gap/2⌋
14:    end while
15: end procedure
```

0.3.5. Árbol binario de búsqueda

Algorithm 7 Árbol binario de búsqueda

```
1: procedure InsertBST(Arbol, elemento)
2:     posicion ← Arbol
3:     while posicion ≠ NULL do
4:         if elemento < posicion.elemento then
5:             posicion ← posicion.izquierda
6:         else
7:             posicion ← posicion.derecha
8:         end if
9:     end while
10:    posicion ← NuevoNodo(elemento)
11: end procedure
12: procedure Inorder(Arbol, A, n)
13:    posicion ← Arbol
14:    i ← 0
15:    pila ← {}
16:    do
17:        while posicion ≠ NULL do
18:            pila.push(posicion)
19:            posicion ← posicion.izquierda
20:        end while
21:        if pila es no vacía then
22:            posicion ← pila.pop()
23:            A[i] ← posicion.elemento
24:            i ← i + 1
25:            posicion ← posicion.derecha
26:        end if
27:    while posicion ≠ NULL y pila es no vacía
28: end procedure
29: procedure SortWithBST(A, n)
30:    Arbol ← NULL
31:    for i ← 0 hasta n − 1 do
32:        InsertBST(Arbol, A[i])
33:    end for
34:    Inorder(Arbol, A, n)
35: end procedure
```

0.4. Implementación de la solución

0.4.1. Burbuja

```
k+ktvoid n+nfBubbleSortv1p(k+ktint nDatap[], k+ktint nDataSizep) p
 ↪  c+c1//=== BUBBLE SORT ======
   kfor p(k+ktint ni o= l+m+mi0p; ni o nDataSize o l+m+mi1p; o++nip) p
 ↪  c+c1//Do this Size  1 times
      kfor p(k+ktint nj o= l+m+mi0p; nj o nDataSize o l+m+mi1p; o++njp) p
 ↪  c+c1//Do this Size  1 times
          kif p(nDatap[njp] o nDatap[nj o+ l+m+mi1p])
 ↪  c+c1//If we need to swap it
              nSwapp(onDatap[njp], onDatap[nj o+ l+m+mi1p]);
 ↪  c+c1//Swap it!
      p
   p
p
```

```
k+ktvoid n+nfBubbleSortv2p(k+ktint nDatap[], k+ktint nDataSizep) p
   kfor p(k+ktint ni o= l+m+mi0p; ni o nDataSize o l+m+mi1p; nio++p) p
 ↪  c+c1//Do this Size  1 times
      kfor p(k+ktint nj o= l+m+mi0p; nj o nDataSize o ni o l+m+mi1p; njo++p) p
 ↪  c+c1//Last i are sorted
          kif p(nDatap[njp] o nDatap[nj o+ l+m+mi1p])
 ↪  c+c1//If we need to swap it
              nSwapp(onDatap[njp], onDatap[nj o+ l+m+mi1p]);
 ↪  c+c1//Swap it!
      p
   p
p
```

```
k+ktvoid n+nfBubbleSortv3p(k+ktint nDatap[], k+ktint nDataSizep) p

   kfor p(k+ktint ni o= l+m+mi0p; ni o nDataSize o l+m+mi1p; nio++p) p
 ↪  c+c1//Do: Size  1 times
      k+ktbool nSwapped o= n+nbfalsep;
 ↪  c+c1//Suposse no swap

      kfor p(k+ktint nj o= l+m+mi0p; nj o nDataSize o ni o l+m+mi1p; njo++p) p
 ↪  c+c1//Last i are sorted
          kif p(nDatap[njp] o nDatap[nj o+ l+m+mi1p]) p
 ↪  c+c1//We need to swap it?
              nSwapp(onDatap[njp], onDatap[nj o+ l+m+mi1p]);
 ↪  c+c1//Swap it!
              nSwapped o= n+nbtruep;
 ↪  c+c1//Upps! swap
          p
      p

```

```
13          kif p(o!nSwappedp) kbreakp;                                    c+c1//No
    ↪  swap,Its sorted!
14       p
15  p
```

0.4.2. Inserción

```
1   k+ktvoid n+nfInsertionSortp(k+ktint nDatap[], k+ktint nDataSizep)
    ↪  c+c1//= INSERTION SORT ===
2
3       kfor p(k+ktint ni o= l+m+mi1p; ni o nDataSizep; o++nip) p
    ↪  c+c1//Traverse each item
4
5           k+ktint nj o= nip;
    ↪  c+c1//A[0..i1] is already sorted
6           k+ktint nTemp o= nDatap[nip];
    ↪  c+c1//Lets find next item
7
8           kwhile p(nj o l+m+mi0 o nTemp o nDatap[nj o l+m+mi1p]) p
    ↪  c+c1//Move elements of the subarray
9               nDatap[njp] o= nDatap[nj o l+m+mi1p];
    ↪  c+c1//to one element ahead
10              njop;                                                  c+c1//until we
    ↪  find the correct place
11          p
12
13          nDatap[njp] o= nTempp;
    ↪  c+c1//Put there the Temp
14      p
15  p
```

0.4.3. Selección

```
1   k+ktvoid n+nfSelectionSortp(k+ktint nDatap[], k+ktint nDataSizep)
    ↪  c+c1//== SELECTION SORT =
2
3       kfor p(k+ktint ni o= l+m+mi0p; ni o nDataSize o l+m+mi1p; o++nip) p
    ↪  c+c1//For each index
4           k+ktint nTiniestIndex o= nip;
    ↪  c+c1//Save actual index
5           kfor p(k+ktint nj o= ni o+ l+m+mi1p; nj o nDataSizep; o++njp)
    ↪  c+c1//Check if is tinest
6               kif p(nDatap[njp] o nDatap[nTiniestIndexp])
    ↪  c+c1//If exists smallest
7                   nTiniestIndex o= njp;
    ↪  c+c1//Change the index
8
9           nSwapp(onDatap[nTiniestIndexp], onDatap[nip]);
    ↪  c+c1//Swap tiniest data
```

```
10      p
11  p
```

0.4.4. Shell

```
1   k+ktvoid n+nfShellSortp(k+ktint nDatap[], k+ktint nDataSizep) p
     ↪  c+c1//=== SHELL SORT ======
2       k+ktint nGap o= nDataSize o l+m+mi1p;
     ↪  c+c1//Let jump = DataSize / 2
3
4       kwhile p(nGap o l+m+mi0p) p
     ↪  c+c1//Until entire data sort
5
6           kforp(k+ktint ni o= nGapp; ni o nDataSizep; nio++p) p
     ↪  c+c1//For each subarray
7
8               k+ktint nj o= nip;
     ↪  c+c1//Let j = i to modify j
9               k+ktint nTemporal o= nDatap[nip];
     ↪  c+c1//Temp. will be the next
10
11              kwhile p(nj o= nGap o nTemporal o nDatap[nj o nGapp]) p
     ↪  c+c1//shift earlier gapsort
12                  nDatap[njp] o= nDatap[nj o nGapp];
     ↪  c+c1//until correct place
13                  nj o= nGapp;                                        c+c1//is
     ↪  found for Data[i]
14              p
15
16              nDatap[njp] o= nTemporalp;
     ↪  c+c1//Temp. find their place
17          p
18
19          nGap o= l+m+mi1p;
     ↪  c+c1//Reduce gap in half
20      p
21  p
```

0.4.5. Árbol binario de búsqueda

```
1   ktypedef kstruct nNode p                                            c+c1//===
     ↪  NODE ===
2     k+ktint  nNodeItemp;
     ↪  c+c1//Pointer to the real data
3     kstruct nNode o*nLeftp;
     ↪  c+c1//Pointer to the left node
4     kstruct nNode o*nRightp;
     ↪  c+c1//Pointer to the left node
```

```
5  p nNodep;                                                          c+c1//We call
    ↪  this struct a node
6
7  ktypedef nNode nBinaryTreep;                                        c+c1//New
    ↪  name same functionality
```

```
1  kextern k+krinline nNodeo* n+nfCreateBinaryTreep(k+ktint nNewItemp) p
    ↪  c+c1// ==== CREATE A NEW NODE ==
2     nNode o*nNewNode o= p(nNodeo*p) nmallocp(ksizeofp(nNodep));
    ↪  c+c1//Reserve memory for  node
3     nNewNodeonNodeItem o= nNewItemp;                                 c+c1//You
    ↪  will protect this
4     nNewNodeonLeft o= n+nbNULLp;
    ↪  c+c1//And maybe youre a leaf
5     nNewNodeonRight o= n+nbNULLp;
    ↪  c+c1//And maybe youre a leaf
6     kreturn nNewNodep;                                              c+c1//Go, go
    ↪  NewNode :)
7  p
```

```
1  k+ktvoid n+nfIterativeInsertBSTp(nBinaryTree o**nTreep, k+ktint nNewItemp) p
    ↪  c+c1// ==== INSERT IN A TREE ==
2     nNode o**nNewNode o= nTreep;                                      c+c1//Let
    ↪  start at root
3
4     kwhile p(o*nNewNode o!= n+nbNULLp)
    ↪  c+c1//While are not at a leaf
5        nNewNode o= p(nNewItem o p(o*nNewNodep)onNodeItemp)o?
    ↪  c+c1//We have to move right
6           op((o*nNewNodep)onLeftp)o: op((o*nNewNodep)onRightp);
    ↪  c+c1//Move left or right
7
8     p(o*nNewNodep) o= nCreateBinaryTreep(nNewItemp);
    ↪  c+c1//Create a node at a leaf
9  p
```

```
1  k+ktvoid n+nfIterativeCreateInOrderp(nNode o**nTreep, k+ktint o*nDatap, k+ktint
    ↪  nDataSizep) p c+c1//= CREATE ARRAY FROM TREE ==
2     nNode o*nActualNode o= o*nTreep;
    ↪  c+c1//Now, use a temporal node
3
4     nNode o**nStack o= ncallocp(nDataSizep, ksizeofp(nNodeo*p));
    ↪  c+c1//Create a stack like
5
6     k+ktint nStackPointer o= ol+m+mi1p, ni o= l+m+mi0p;
    ↪  c+c1//Aux variables
7
8     kdo p                                                           c+c1//Do the
    ↪  next:
9
```

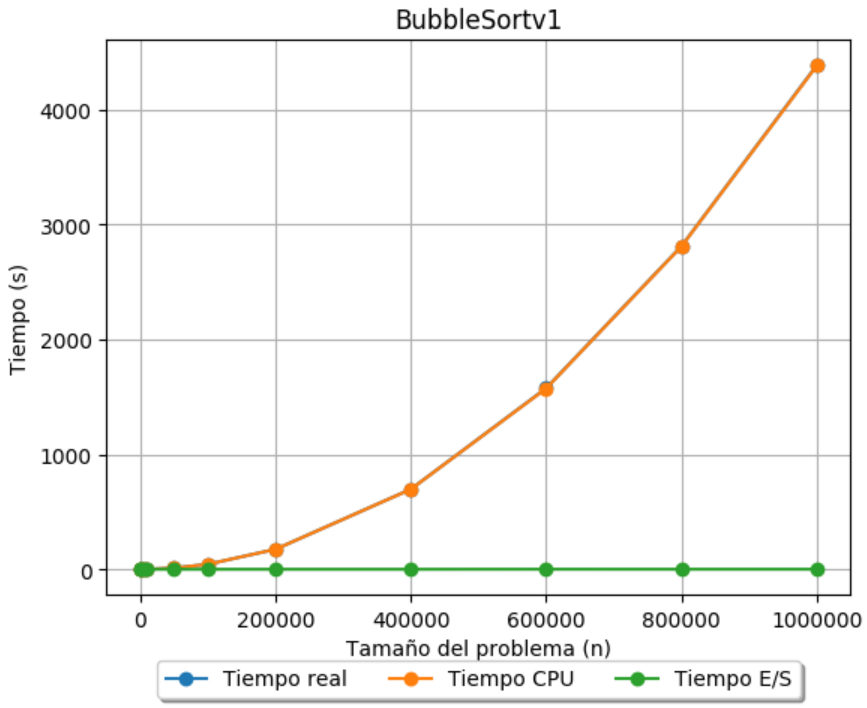
```
10          kwhile p(nActualNode o!= n+nbNULLp) p
    ↪   c+c1//While we are not a leaf
11              nStackp[o++nStackPointerp] o= nActualNodep;
    ↪   c+c1//Add to stack the node
12              nActualNode o= nActualNodeonLeftp;                          c+c1//Move
    ↪   to the fuck to left
13          p
14
15          kif p(nStackPointer o= l+m+mi0p) p
    ↪   c+c1//Ok, now while not empty
16              nActualNode o= nStackp[nStackPointerop];
    ↪   c+c1//Start pushing the data
17              nDatap[nio++p] o= nActualNodeonNodeItemp;
    ↪   c+c1//Now, add to the data array
18              nActualNode o= nActualNodeonRightp;                         c+c1//And
    ↪   move to right
19          p
20      p
21      kwhile p(nActualNode o!= n+nbNULL o|| nStackPointer o= l+m+mi0p);
    ↪   c+c1//Now do this while we can
22
23      nfreep(nStackp);                                                    c+c1//Bye
    ↪   Stack :)
24  p
```

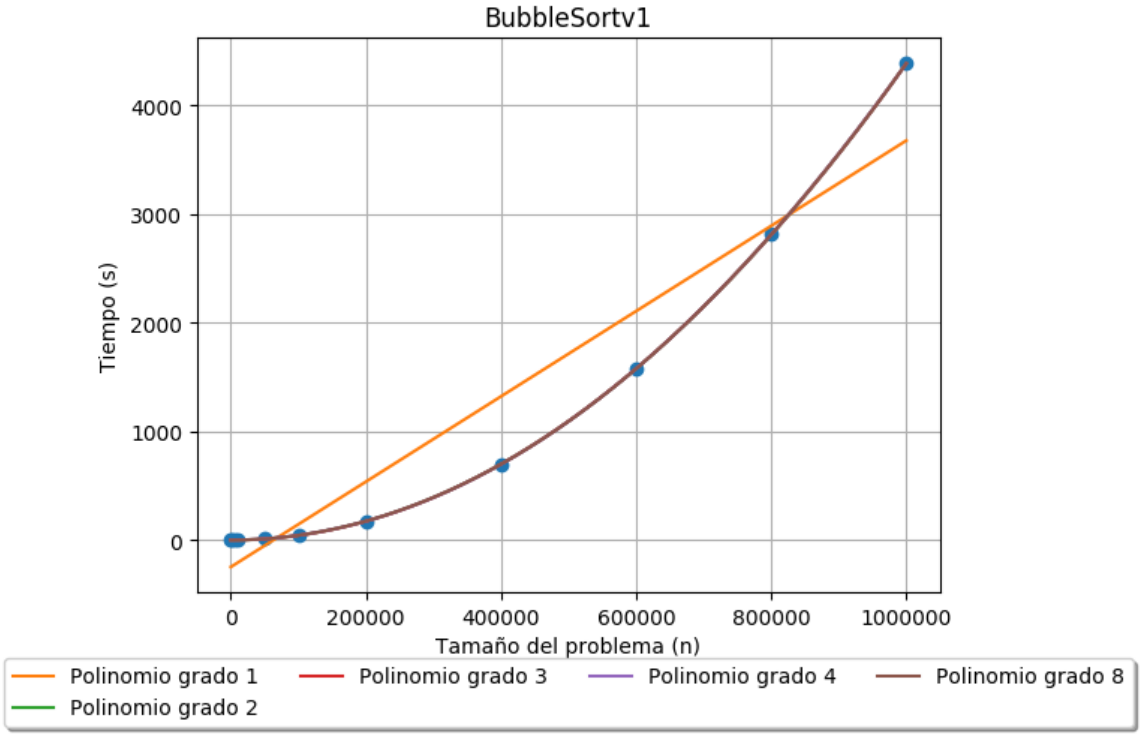
```
1   k+ktvoid n+nfSortWithBSTp(k+ktint nDatap[], k+ktint nDataSizep) p
    ↪   c+c1//=== BST SORT ======
2       nNodeo* nTree o= n+nbNULLp;
    ↪   c+c1//Start with empty tree
3
4       kforp(k+ktint ni o= l+m+mi0p; ni o nDataSizep; nio++p)
    ↪   c+c1//For each element
5           nIterativeInsertBSTp(onTreep, nDatap[nip]);
    ↪   c+c1//Insert in tree
6
7       nIterativeCreateInOrderp(onTreep, nDatap, nDataSizep);
    ↪   c+c1//Transverse it!
8   p
```

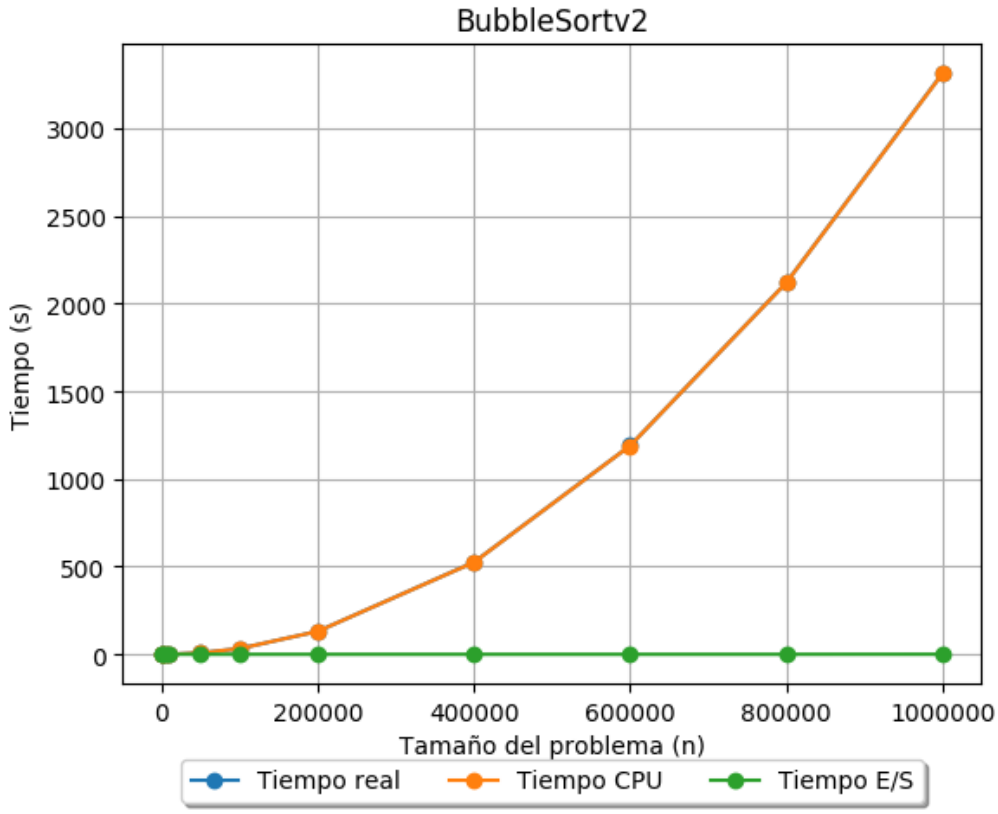
0.5. Actividades y pruebas

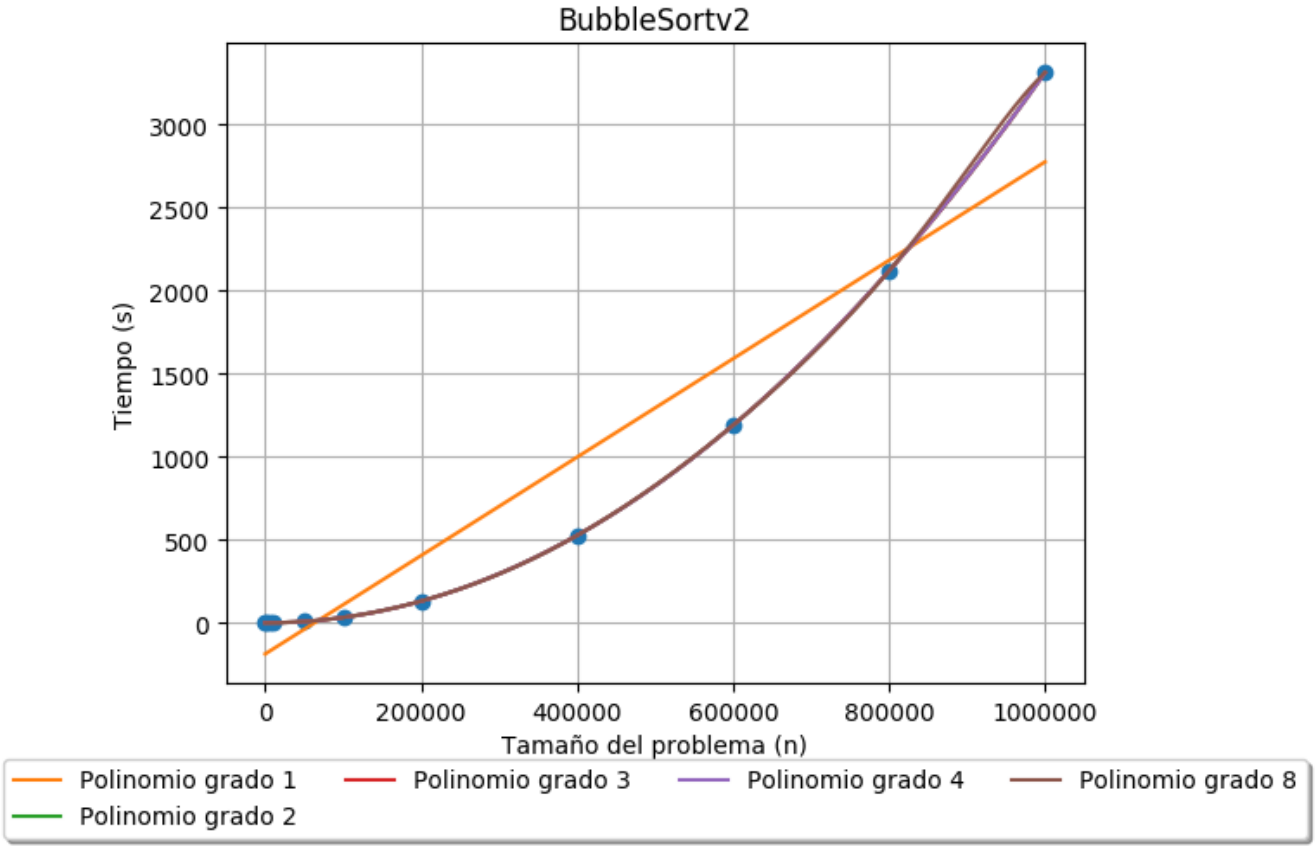
0.5.1. Comparativas individuales

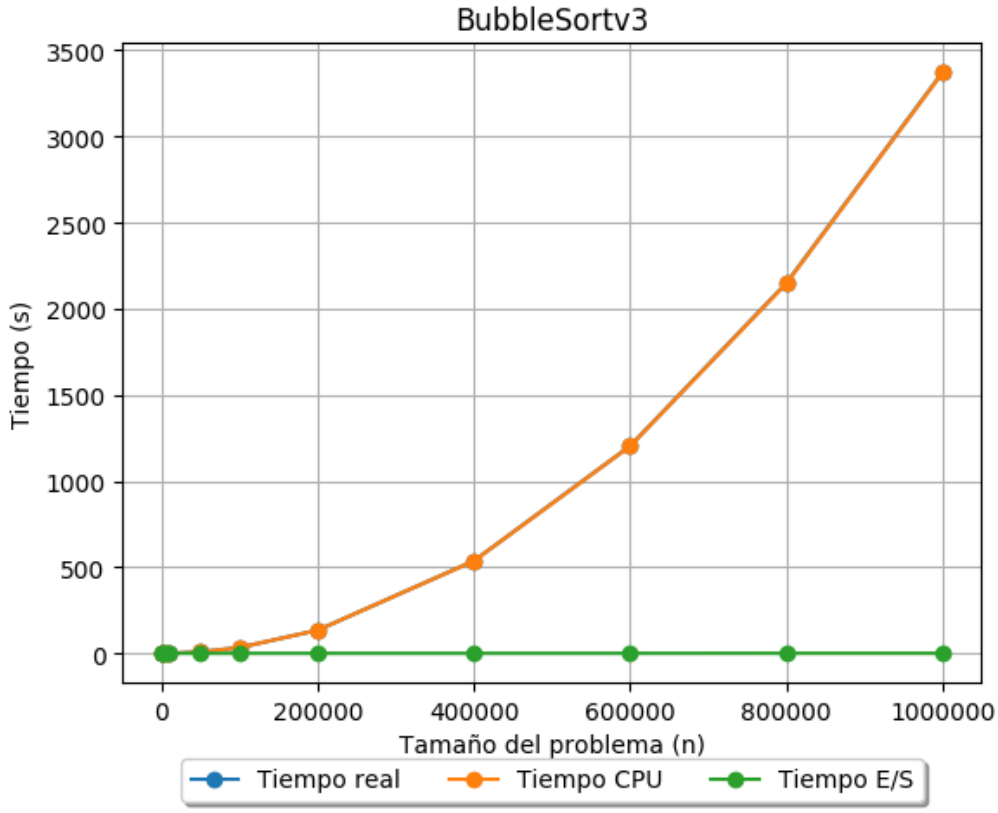
Burbuja

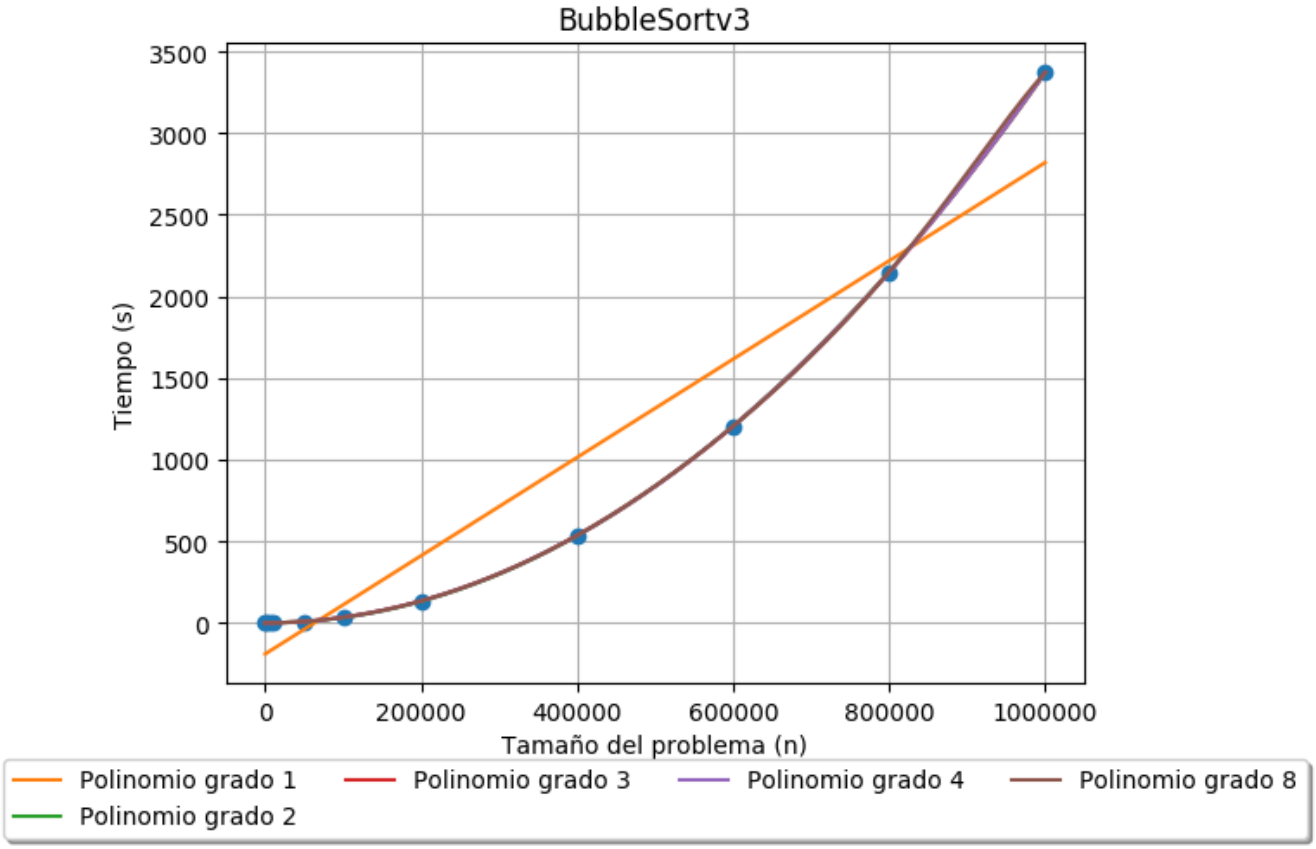



BubbleSortv1
Tiempo (s)
4000
3000
2000
1000
0
0
200000
400000
600000
800000
1000000
Tamaño del problema (n)
Polinomio grado 1
Polinomio grado 2
Polinomio grado 3
Polinomio grado 4
Polinomio grado 8

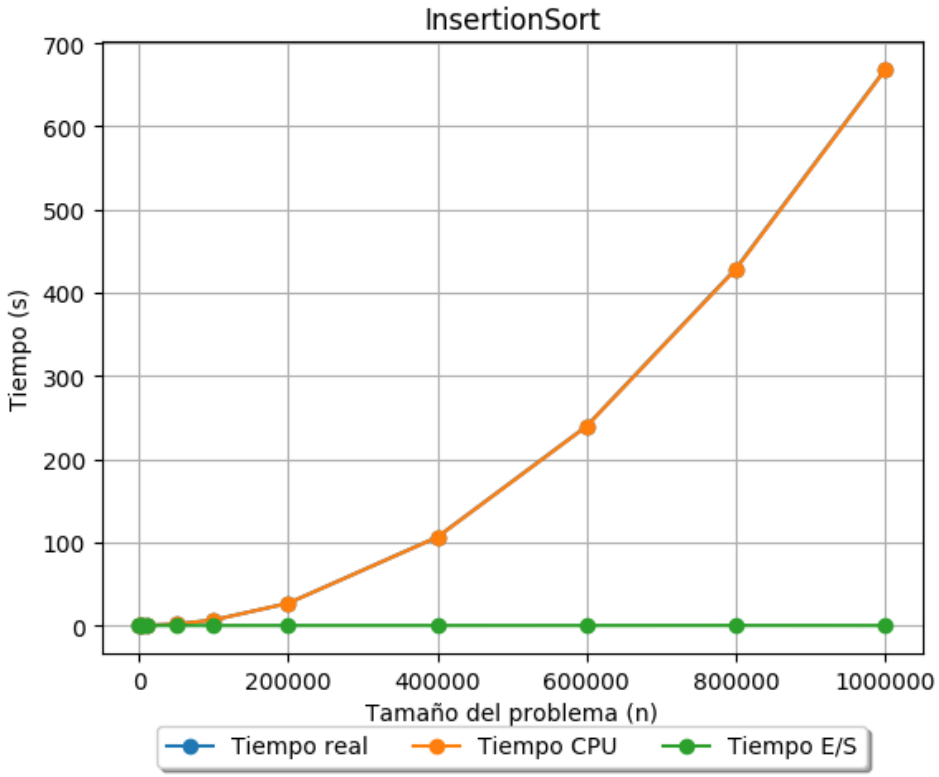

BubbleSortv2
Tiempo (s)
3000
2500
2000
1500
1000
500
0
0
200000
400000
600000
800000
1000000
Tamaño del problema (n)
Tiempo real
Tiempo CPU
Tiempo E/S

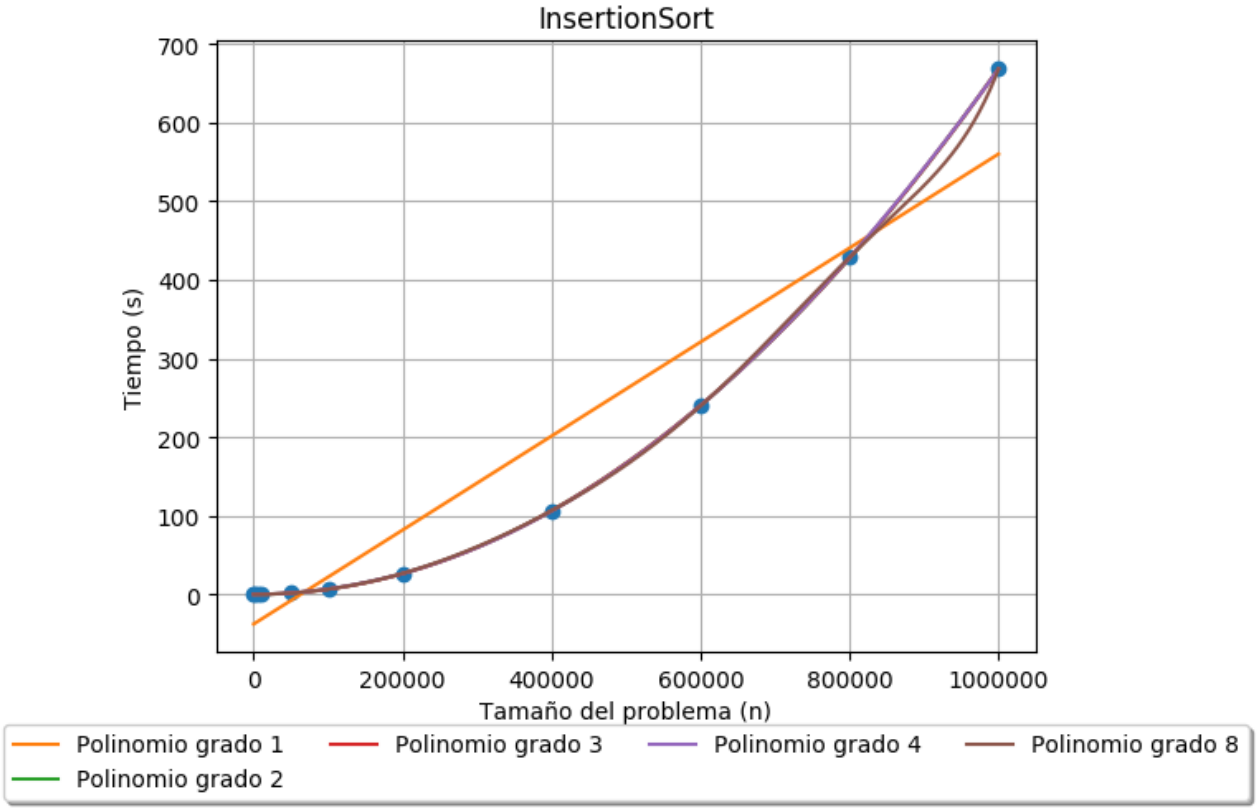

BubbleSortv2
Tiempo (s)
3000
2500
2000
1500
1000
500
0
0
200000
400000
600000
800000
1000000
Tamaño del problema (n)
Polinomio grado 1
Polinomio grado 2
Polinomio grado 3
Polinomio grado 4
Polinomio grado 8


BubbleSortv3
Tiempo (s)
3500
3000
2500
2000
1500
1000
500
0
0
200000
400000
600000
800000
1000000
Tamaño del problema (n)
Tiempo real
Tiempo CPU
Tiempo E/S

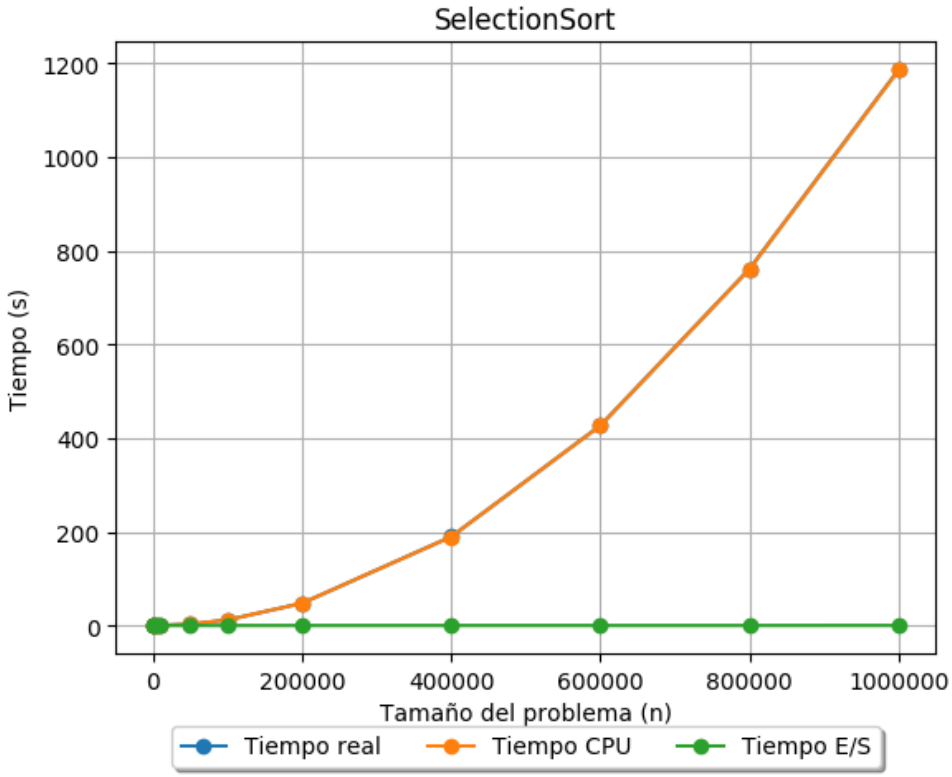

BubbleSortv3
Tiempo (s)
Tamaño del problema (n)
0
500
1000
1500
2000
2500
3000
3500
0
200000
400000
600000
800000
1000000
Polinomio grado 1
Polinomio grado 2
Polinomio grado 3
Polinomio grado 4
Polinomio grado 8

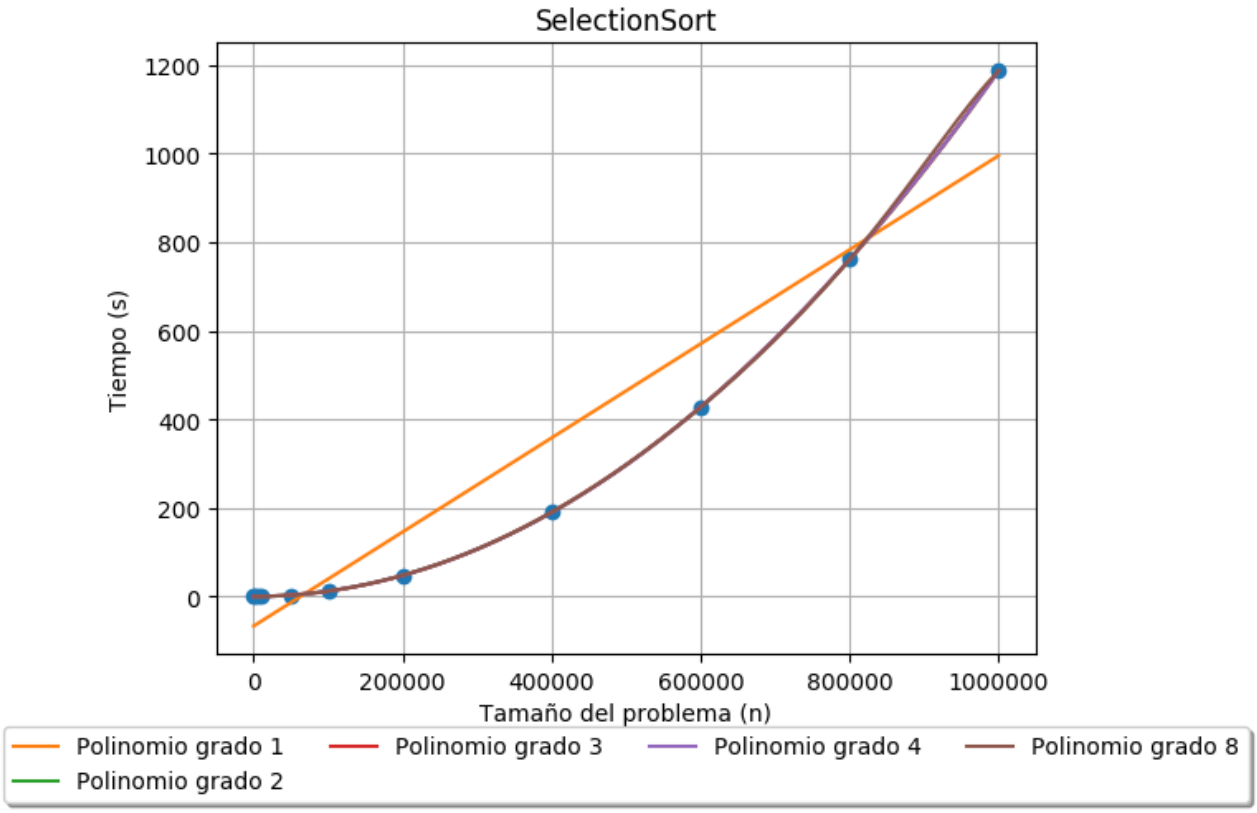
Inserción


InsertionSort
Tiempo (s)
700
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1000000
Tamaño del problema (n)
Tiempo real
Tiempo CPU
Tiempo E/S

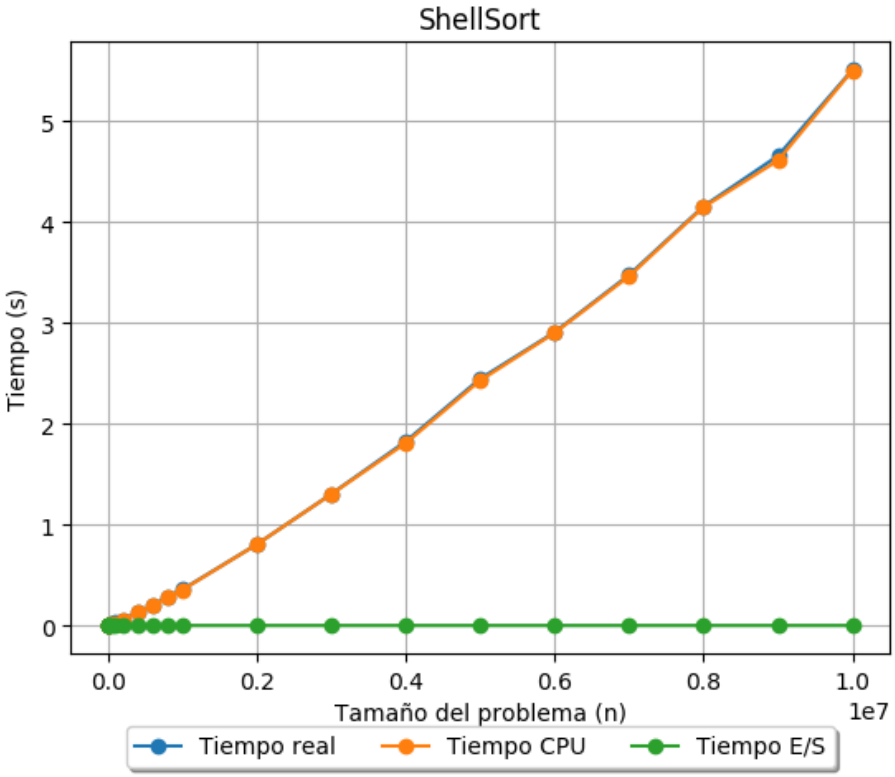

InsertionSort
Tiempo (s)
Tamaño del problema (n)
0
100
200
300
400
500
600
700
0
200000
400000
600000
800000
1000000
Polinomio grado 1
Polinomio grado 2
Polinomio grado 3
Polinomio grado 4
Polinomio grado 8

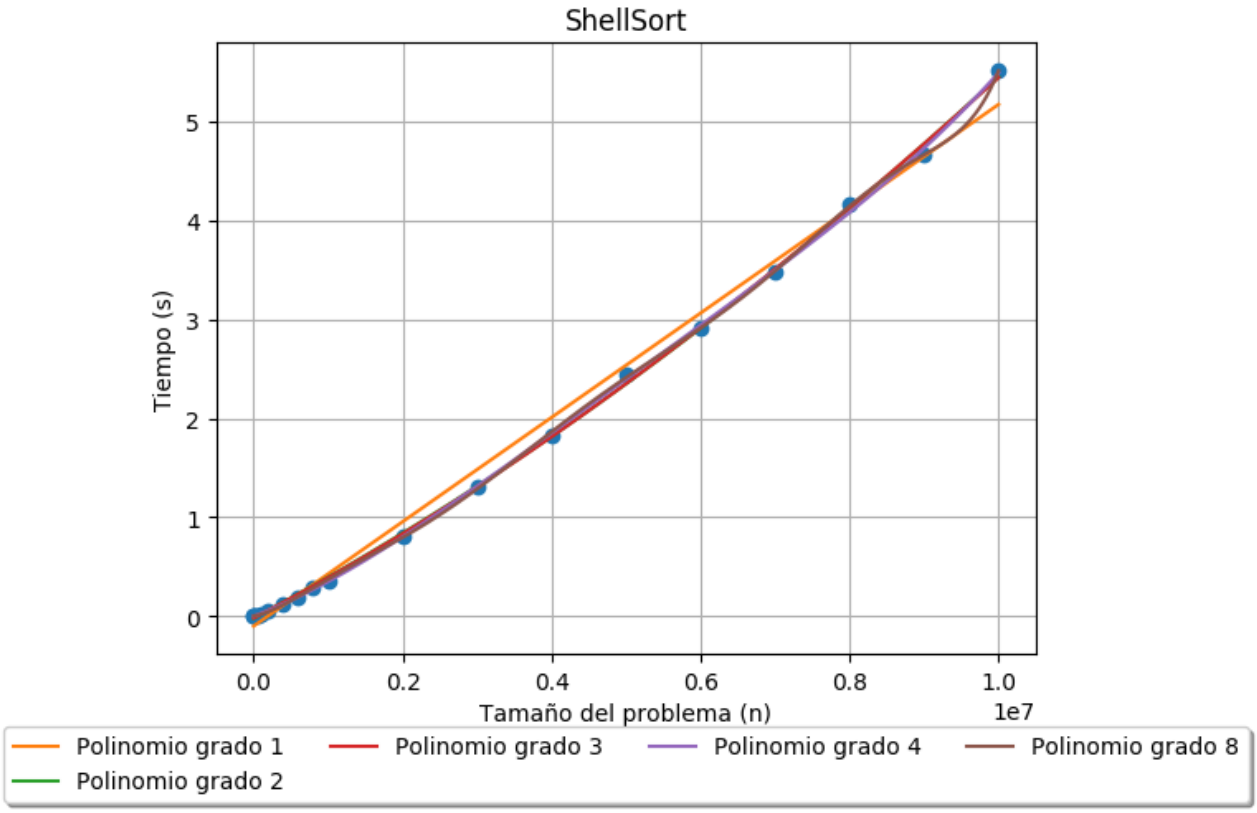
Selección


SelectionSort
Tiempo (s)
1200
1000
800
600
400
200
0
0
200000
400000
600000
800000
1000000
Tamaño del problema (n)
Tiempo real
Tiempo CPU
Tiempo E/S

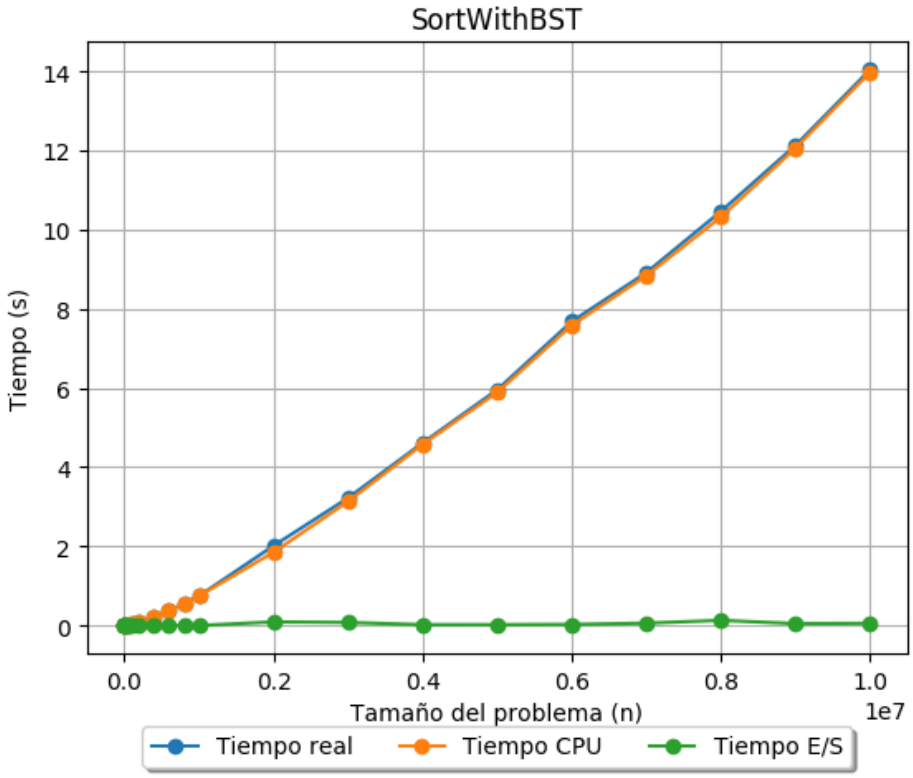

SelectionSort
1200
1000
800
600
400
200
0
Tiempo (s)
0
200000
400000
600000
800000
1000000
Tamaño del problema (n)
Polinomio grado 1
Polinomio grado 2
Polinomio grado 3
Polinomio grado 4
Polinomio grado 8

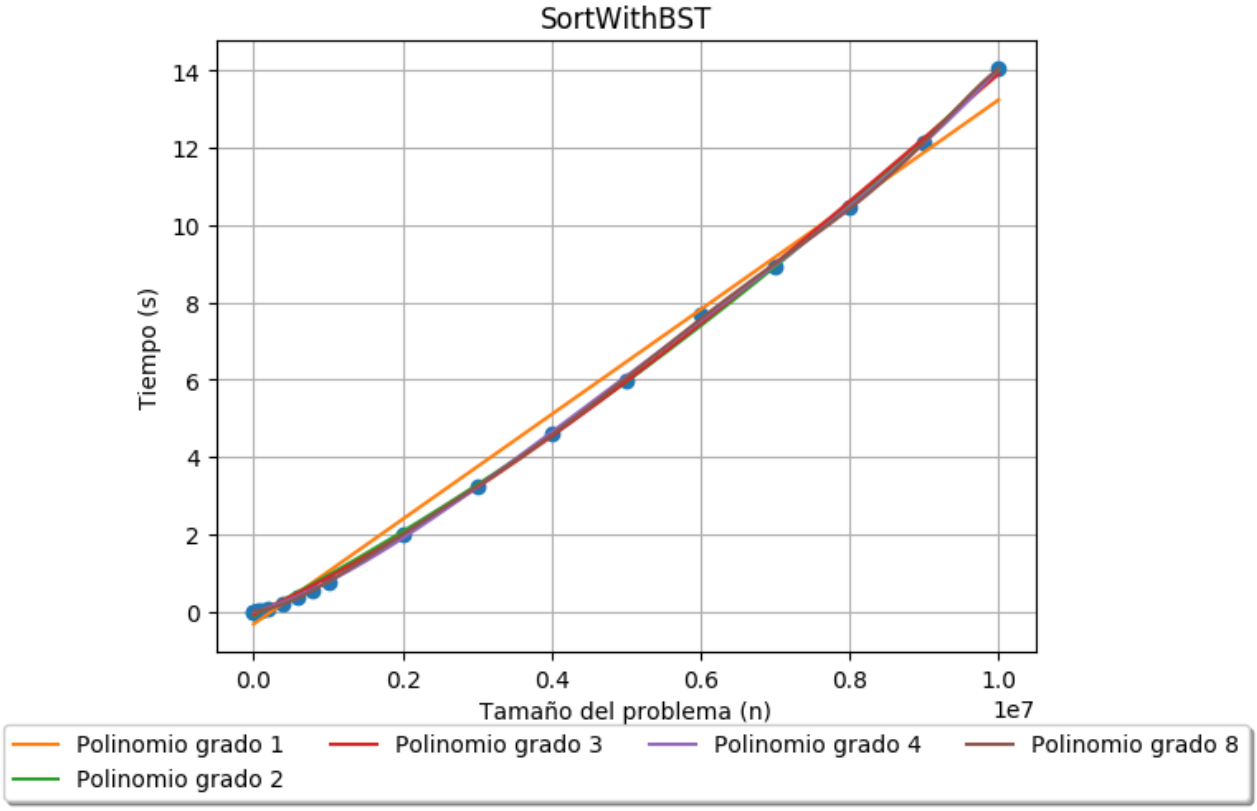
Shell


ShellSort
Tiempo (s)
0
1
2
3
4
5
0.0
0.2
0.4
0.6
0.8
1.0
1e7
Tamaño del problema (n)
Tiempo real
Tiempo CPU
Tiempo E/S


ShellSort
Tiempo (s)
0
1
2
3
4
5
0.0
0.2
0.4
0.6
0.8
1.0
1e7
Tamaño del problema (n)
Polinomio grado 1
Polinomio grado 2
Polinomio grado 3
Polinomio grado 4
Polinomio grado 8

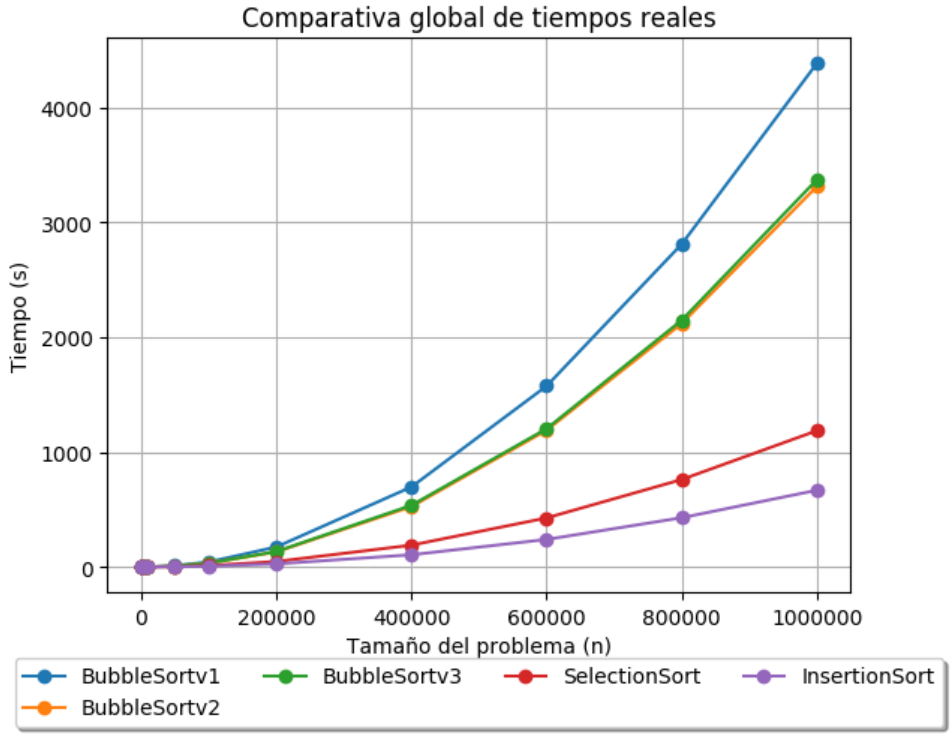
Árbol binario de búsqueda

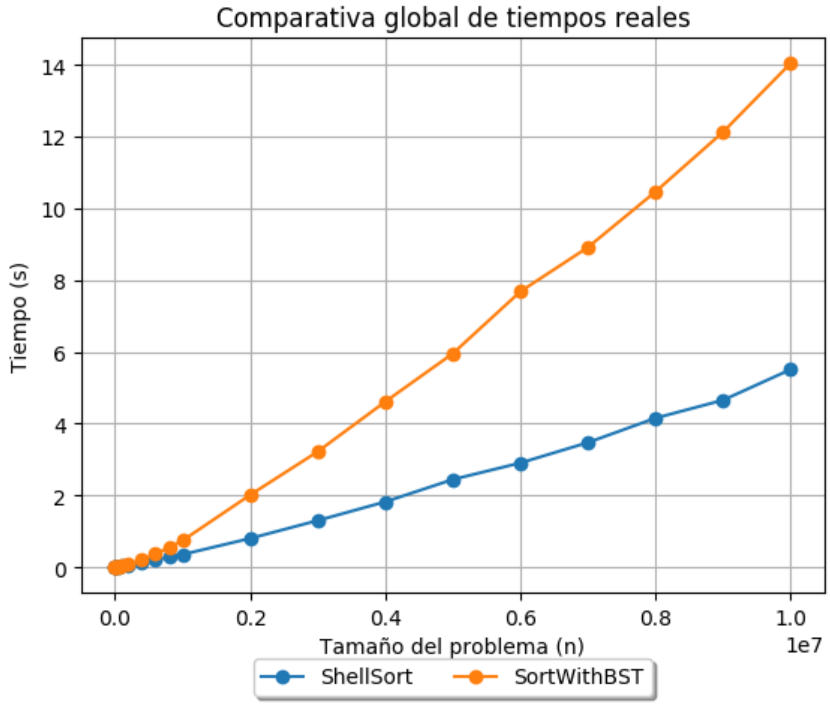


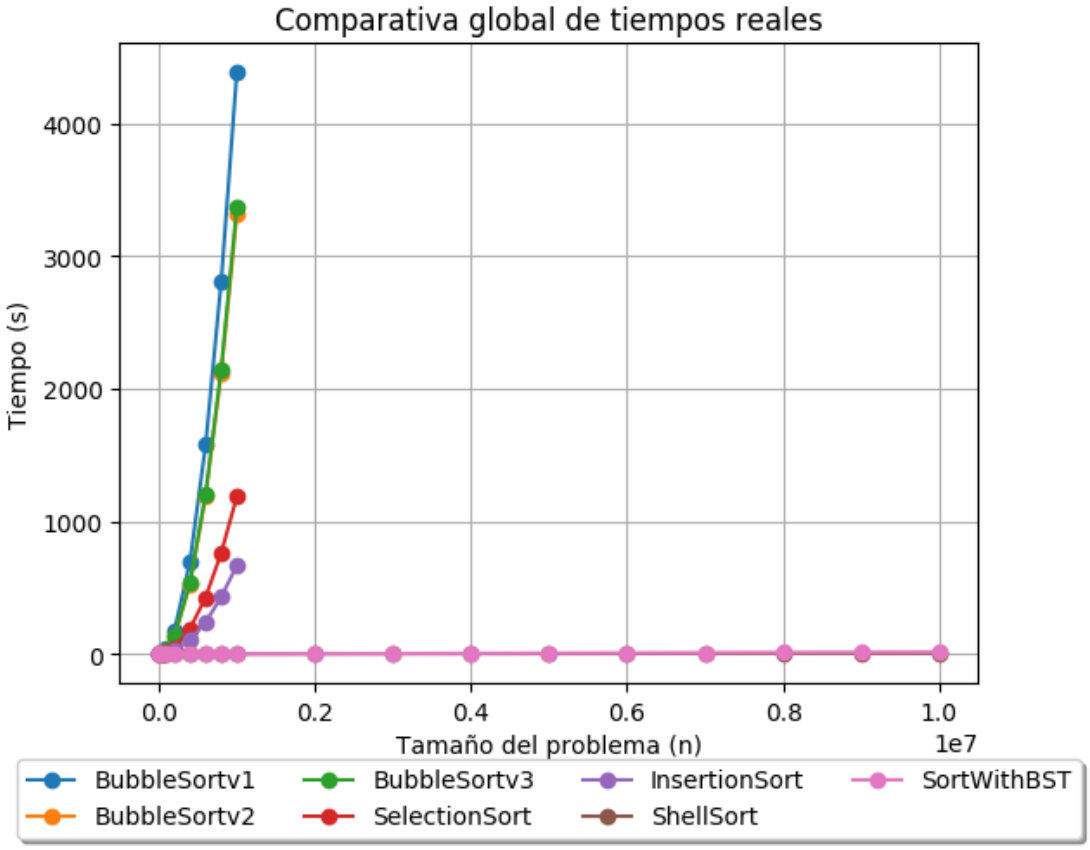

SortWithBST
Tiempo (s)
Tamaño del problema (n)
1e7
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12
14
Polinomio grado 1
Polinomio grado 2
Polinomio grado 3
Polinomio grado 4
Polinomio grado 8

0.5.2. Comparativas globales

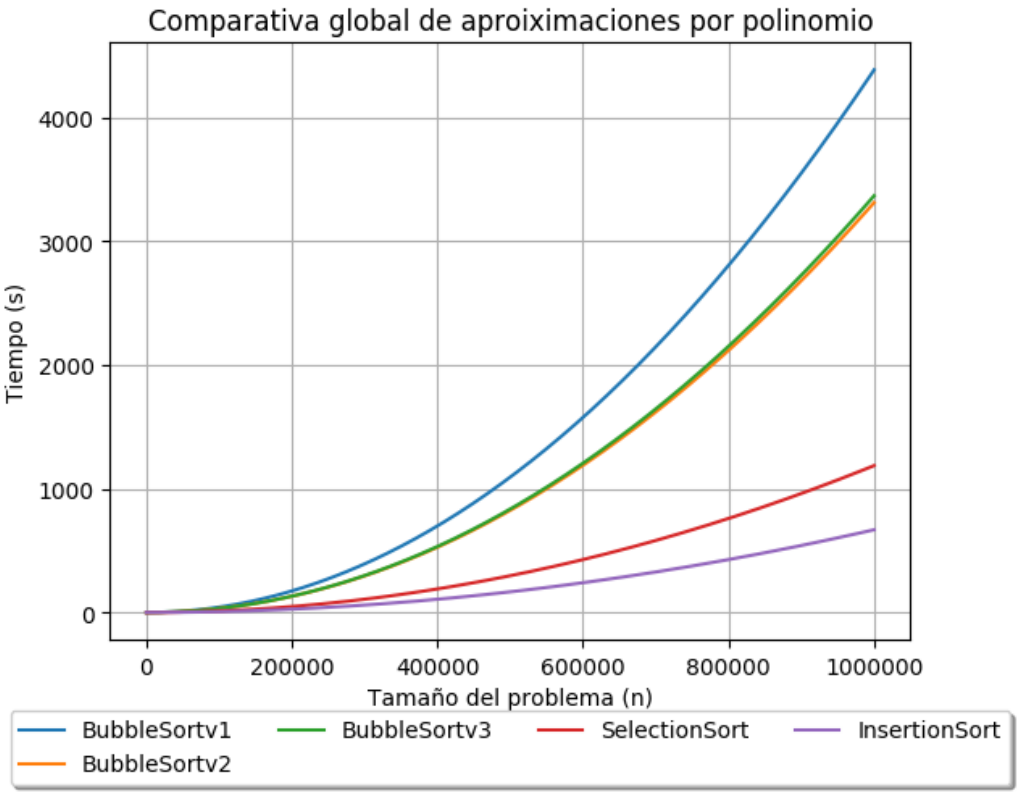
Por tiempo real

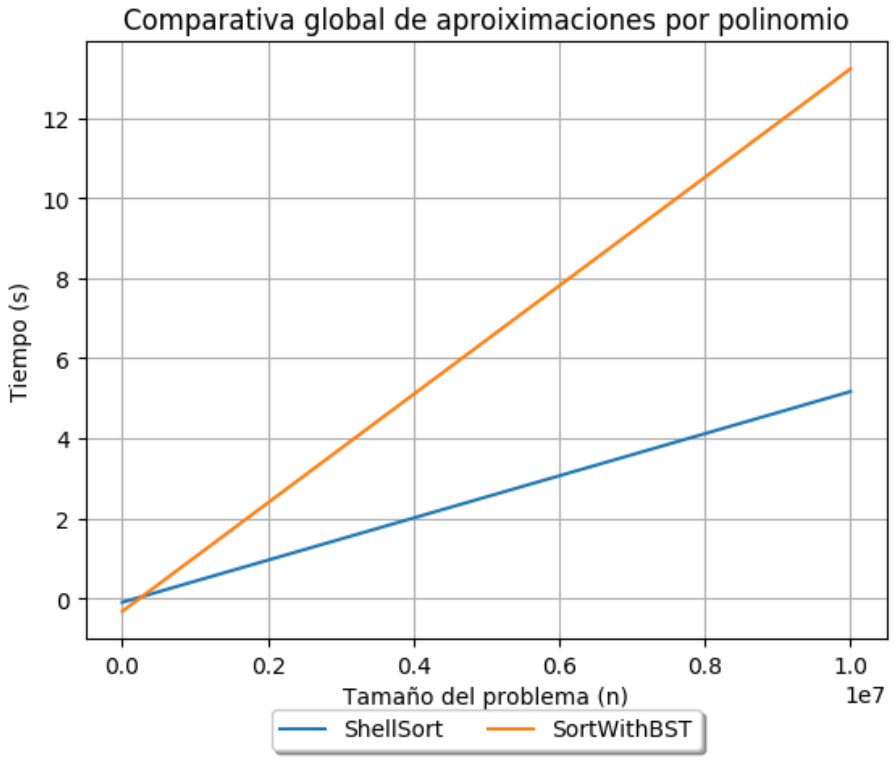


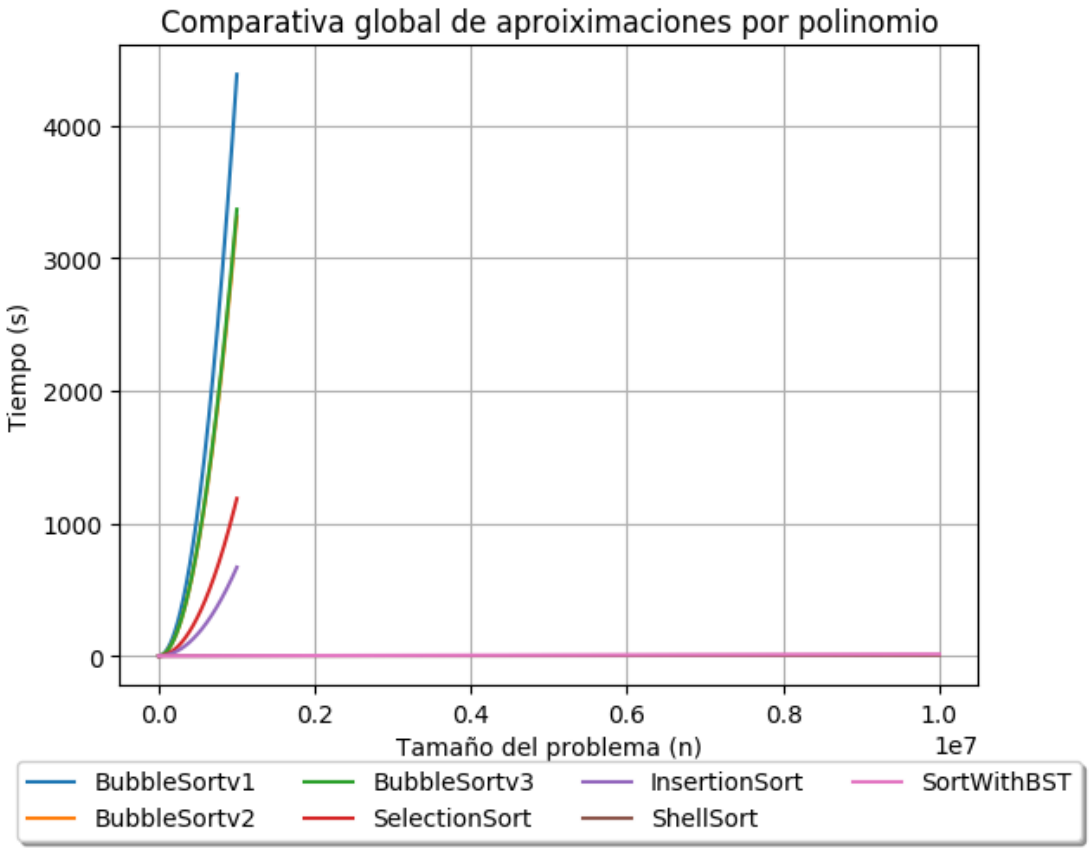

Comparativa global de tiempos reales
Tiempo (s)
0
2
4
6
8
10
12
14
0.0
0.2
0.4
0.6
0.8
1.0
1e7
Tamaño del problema (n)
ShellSort
SortWithBST


Comparativa global de tiempos reales
Tiempo (s)
0
1000
2000
3000
4000
0.0
0.2
0.4
0.6
0.8
1.0
1e7
Tamaño del problema (n)
BubbleSortv1
BubbleSortv2
BubbleSortv3
SelectionSort
InsertionSort
ShellSort
SortWithBST

Por polinomio




Comparativa global de aproximaciones por polinomio
Tiempo (s)
0
2
4
6
8
10
12
0.0
0.2
0.4
0.6
0.8
1.0
1e7
Tamaño del problema (n)
ShellSort
SortWithBST



Los polinomios escogidos como mejores aproximaciones fueron:

- BubbleSort v1:
  $P(n) = 4.39614394 \times 10^{-9}n^2 - 9.38843814 \times 10^{-6}n + 0.06536545$
- BubbleSort v2:
  $P(n) = 3.32645251 \times 10^{-9}n^2 - 1.30521594 \times 10^{-5}n + 0.19869421$
- BubbleSort v3:
  $P(n) = 3.39403986 \times 10^{-9}n^2 - 2.61286379 \times 10^{-5}n + 1.00078946$
- SelectionSort:
  $P(n) = 1.18611115 \times 10^{-9}n^2 + 1.05506770 \times 10^{-6}n - 0.02919735$
- InsertionSort:
  $P(n) = 6.70213950 \times 10^{-10}n^2 - 1.77898112 \times 10^{-6}n + 0.04638817$
- ShellSort:
  $P(n) = 5.26723624 \times 10^{-7}n - 0.09826119$

- SortWithBST:

  $P(n) = 1.35578423 \times 10^{-6}n - 0.32358203$

Para los primeros 5 algoritmos escogimos un polinomio cuadrático, pues es lo más cercano a la complejidad del caso medio, $O(n^2)$.

Para los últimos dos usamos un polinomio lineal, pues es el más cercano a la complejidad teórica de $O(n \log n)$ en el caso medio.

Si usamos esos polinomios para estimar los tiempos para valores de n mucho más grandes, obtenemos los siguientes tiempos estimados:

	$n = 5 \times 10^{-7}$	$n = 10^8$	$n = 5 \times 10^8$	$n = 10^9$	$n = 5 \times 10$
BubbleSortv1	10,989,890.5 s	43,960,500.6 s	1,099,031,292.4 s	4,396,134,557.8 s	109,903,551,71
BubbleSortv2	8,315,478.8 s	33,263,220.1 s	831,606,602.4 s	3,326,439,461.2 s	83,161,247,57
BubbleSortv3	8,483,794.2 s	33,937,786.7 s	848,496,902.9 s	3,394,013,737.5 s	84,850,865,98
SelectionSort	2,965,330.6 s	11,861,217.0 s	296,528,315.7 s	1,186,112,208.2 s	29,652,784,10
InsertionSort	1,675,445.9 s	6,701,961.6 s	167,552,598.0 s	670,212,171.0 s	16,755,339,85
ShellSort	26.2 s	52.5 s	263.2 s	526.6 s	2,633.5 s
SortWithBST	67.4 s	135.2 s	677.5 s	1,355.4 s	6,778.5 s

0.5.3. Preguntas

1. **¿Cuál de los 5 algoritmos es más fácil de implementar?**

   El algoritmo de burbuja en sus tres versiones, porque solo requerimos de realizar intercambios entre elementos consecutivos del arreglo, lo cual es relativamente sencillo.

2. **¿Cuál de los 5 algoritmos es el más difícil de implementar?**

   El algoritmo con el árbol binario de búsqueda, porque tuvimos que trabajar con punteros, crear una estructura para los nodos, y convertir los procedimientos de inserción y recorrido inorden a sus versiones iterativas.

3. **¿Cuál algoritmo tiene menor complejidad temporal?**

   Teóricamente el árbol binario de búsqueda, pues en este caso como los números están distribuidos uniformemente, el árbol creado estará más o menos balanceado, logrando la inserción de cada elemento en $O(\log n)$ y la inserción de todos en $O(n \log n)$.

   Sin embargo, el más rápido en la práctica fue Shell Sort, tomando la mitad de tiempo que el árbol binario de búsqueda. Los saltos escogidos sirvieron para reducir el número de inserciones.

4. **¿Cuál algoritmo tiene mayor complejidad temporal?**

   BubbleSort versión 1, porque siempre realiza $O(n^2)$ intercambios, sin importar cómo estén distribuidos los elementos del arreglo.

5. **¿Cuál algoritmo tiene menor complejidad espacial? ¿Por qué?**

   InsertionSort y ShellSort, porque el ordenamiento de los elementos se realiza sobre el mismo arreglo, y como aquí no se realizan intercambios sino solo desplazamientos, no necesitamos memoria extra.

   Aunque BubbleSort y SelectionSort también realizan el ordenamiento sobre el arreglo, requieren un poco más de memoria para intercambiar elementos, pero la diferencia con los anteriores es casi nula.

6. **¿Cuál algoritmo tiene mayor complejidad espacial? ¿Por qué?**

   El árbol binario de búsqueda, porque necesitamos construir el árbol, cuya cantidad de nodos es la misma que la de los elementos del arreglo, requiriendo el doble de memoria para ordenarlo. Además, como se usó la versión iterativa del recorrido inorden, tuvimos que hacer una pila para simular la recursión, y como esa pila guarda todos los nodos del árbol, en total usamos el triple de memoria.

7. **¿El comportamiento experimental de los algoritmos era el esperado? ¿Por qué?**

   - El de BubbleSort versión 1 fue el más tardado de todos, lo cuál esperábamos.

- Las otras dos versiones de BubbleSort tardaron casi lo mismo, pero menos que la versión 1; por lo que la optimización de revisar si ya no habíamos hecho intercambios no fue muy efectiva en este caso.
- El SelectionSort tardó menos, lo cual también esperábamos, aunque también siempre realice $O(n^2)$ operaciones.
- El InsertionSort no estuvo nada mal, siendo el más rápido hasta este punto, pues no realiza intercambios, solo desplazamientos; y es eficiente si el arreglo está más o menos ordenado.
- El ShellSort nos sorprendió, pues fue el más rápido de todos. Al parecer los saltos escogidos fueron los adecuados para reducir drásticamente el tiempo de ordenamiento.
- El ordenamiento con el BST fue el segundo más rápido, lo cuál esperábamos porque, aparte de insertar cada elemento al árbol, hay que recorrerlo y volverlo a copiar al arreglo original. Tardó el doble que ShellSort.

8. **¿Sus resultados experimentales difieren mucho de los del resto de los equipos? ¿A qué se debe?**

   Comprobamos con el equipo Git Gud Team Arbol ,especialmente con Manuel, ejecutamos los códigos gracias al Make.py en la máquina estándar que tiene su equipo y comprobamos que las gráficas son prácticamente iguales, sin importar la implementación el compilador llego básicamente al mismo programa ejecutable.

   Por otro lado si que tenemos diferencias al comprobar los datos cada uno usando sus propios equipos, estas pequeñas diferencias se deben sobretodo a las especificaciones.

9. **¿Existió un entorno controlado para realizar las pruebas experimentales? ¿Cuál fue?**

   Sí, usamos una PC con las siguientes características:

   - HP EliteDesk 700 G1 SFF
   - 8GB de memoria RAM, DDR3 SDRAM non-ECC, 1600MHz
   - Procesador Intel(R) Core(TM) i5-4590 (4° generación), CPU @ 3.30GHz (4 CPUs), ∼3.3GHz. Intel vPro Technology
   - 1TB de disco duro HDD, Serial ATA-600 6Gb/s, 7200 rpm

   Y el software fue el siguiente:

   - Sistema operativo Linux
   - Distribución Elementary OS 0.4
   - Compliador GCC con soporte para C11
   - Python 3.6 con las bibliotecas `matplotlib` y `numpy`
   - No había ninguna aplicación abierta al momento de ejecutar los algoritmos

10. **¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica?**

    - Hagan sus scripts jóvenes, pues aunque parezca más tedioso y crean que es más rápido anotar toda la información solicitada (que es mucha) a mano, es más eficiente por si se requiere cambiar algún algoritmo o los valores de entrada. Al menos que el script guarde los tiempos en algún archivo de texto con un formato que quieran.
    - Prueben sus algoritmos con arreglos pequeños antes de ordenar los 10 millones para ver si realmente los programaron bien.
    - Hagan sus programas generales, es decir, que reciban cualquier tamaño de subarreglo y el algoritmo a usar.
    - Usen Linux, pues en Windows no están las bibliotecas para medir los tres tiempos.
    - Cuando corran sus algoritmos, procuren cerrar todas las tareas en segundo plano para que los tiempos obtenidos sean lo más apegados a la realidad posible.

0.6. Errores detectados

No hemos detectado errores mayores, eso no quiere decir que nuestro software sea completamente perfecto y sin ningun bug, simplemente que no hay ninguno aparente y con los datos con lo que hemos hecho las pruebas, algunos posibles errores teoricos son:

- Usar int como tipo de dato para acceder al array dando la posiblidad a que un arreglo muy grande el mismo indice se desborde

0.7. Posibles mejoras

- Que todos los algoritmos sean capaz de ordenar bajo una relación de orden preestablecida, es decir que nuestros algoritmos reciban una funcion que nos tome 2 elementos y nos diga cual es "mayorçon esto podriamos ordenar de mayor a menor o de menor a mayor con solo cambiar un par de lineas a la funcion que se recibe como parametro
- Que usemos meta programación para no solo arreglar un arreglo de número sino de cualquier tipo de dato comparable, sea por default o por funciones del usuario

0.8. Conclusiones

0.8.1. Alan

En esta práctica revisamos e implementamos algunos ordenamientos de ordenamiento por comparación para una cantidad considerable de elementos a ordenar (10 millones). Estos algoritmos son el ejemplo perfecto para comenzar a estudiar la complejidad temporal, así como la diferencia entre el tamaño del problema y los casos de entrada.

Toda la información recolectada fue experimental y dependió altamente de la plataforma usada y sus recursos, pero de alguna forma sí concuerda con lo que la teoría de cada algoritmo nos dice; por ejemplo, el comportamiento o el orden del algoritmo no deben variar.

BubbleSort fue el peor algoritmo en su primera versión, pues siempre realiza $O(n^2)$ operaciones sin importar cómo estén distribuidos los elementos del arreglo, además está basado en intercambiar únicamente elementos adyacentes, por lo que requiere una variable auxiliar y los elementos no se mueven de forma óptima. Las siguientes dos optimizaciones, basadas en que los últimos elementos van quedando ordenados y verificar si ya no hubo intercambios, bajaron muy poco el tiempo de ejecución.

El siguiente fue SelectionSort, que debido a que siempre tiene que buscar el mínimo, también hará $O(n^2)$ operaciones en todos los casos, pero como el número de intercambios es mucho menor que en BubbleSort, bajó considerablemente el tiempo, pero su orden sigue siendo el mismo.

Luego sigue InsertionSort, que logró casi la mitad de tiempo que SelectionSort, ya que si el arreglo está parcialmente ordenado, haremos todavía menos desplazamientos.

Después tenemos al ordenamiento con el BST, el cual ya fue capaz de ordenar los 10 millones de números completos, a diferencia de los anteriores. Como usa el triple de memoria que el arreglo original, tardó un poco más de lo que esperábamos, pero ya estamos hablando de un algoritmo eficiente.

Finalmente, el ganador fue ShellSort, pues a pesar de ser una simple variante de InsertionSort con orden cuadrático también, el tamaño de los saltos redujo drásticamente el número de desplazamientos, lo cual no siempre sucede.

De esa forma, escogimos polinomios cuadráticos para BubbleSort, SelectionSort e InsertionSort, y polinomios lineales para ShellSort y BST.

0.8.2. Óscar

Gracias a esta practica pudimos entender mucho más a fondo los algoritmos de ordenamiento, primeramente al implementarlo, al pasar de sus respectivos diseños en pseudo-código (o PSeint) a implementaciones en c11, casi todos pudieron ser modela-

dos completamente como una función solo dependiente de sus valores de entrada pero tuvimos además de eso que armar algunas estructuras auxiliares (como BST-árbol binario de búsqueda y un pequeño stack-pila), además a diferencia de las clásicas implementaciones que solemos hacer en estructuras de datos, para evitar una sobrecarga absurda (que podría afectar considerablemente el rendimiento del algoritmo con en sobrecargo de los punteros de funciones) diseñamos los algoritmos para correr de manera iterativa.

Además usamos una característica importante de las versiones modernas de C, como son las funciones inline que nos permiten la modularidad que nos dan las funciones sin tener que perder rendimiento entre los cambios de contexto, pues inline nos permite crear funciones que el compilador puede optimizar de gran manera.

Usamos Python como lenguaje de script para automatizar todo el proceso de corrimiento y de recolección de datos.

Además es importante puntualizar que todas las conclusiones posteriores están bajo los resultados obtenidos, y si bien en cierto que buscamos crear un ambiente de pruebas lo más estable y parejo es importante recordar que estamos tratando con un conjunto, grande si, pero también con una distribución casi normal, con lo cual estamos hablando de que los algoritmos están siendo expuestos y testeados bajo condiciones que no representan como se comportarían bajo una distribución completamente aleatoria, por ejemplo con burbuja llegando a tiempo realmente preocupantes o con un árbol BST completamente desvalanceado por lo que sería de los peores algoritmos o con información casi ordenada lo que nos permitiría ver la gran habilidad de insertion sort para trabajar con un conjunto de números casi ordenados.

Pero dejando en claro estas limitaciones por el tipo de entrada que recibían los algoritmos podemos ver conclusiones muy interesantes, sobre todo hay que tener en cuenta que los ejes que usamos no están acordes, por lo que a primera vista las gráficas pueden dar la apariencia de rectas, pero no nos engañemos, sobretodo al momento de hablar sobre BubbleSort pues su tiempo, y por consiguiente sus operaciones crecen de una manera casi cuadrática con el tamaño del problema.

Puedes notar que con las diversas iteraciones mejora de manera considerable Bubble sort, sí, son números enormes, pero se nota de gran manera cómo es que las modificaciones mejoran de gran manera su rendimiento, hablando en especial de la segunda y tercera versión y cómo una pequeña bandera hace un gran cambio.

Al entrar a hablar de los hermanos, insertion y selection, notamos que ya estamos hablando de tiempo mucho mejores a comparación de las implementaciones de Bubble sort, eso sí con insertionsort siendo casi el doble de rápido, que se dice pronto, pero para algoritmos hermanos no esta nada mal.

Al momento de hablar de BST tenemos la gran ventaja de que está trabajando con datos muy bien distribuidos, por lo tanto incluso sin implementaciones del estilo como AVL podemos tener un árbol bien balanceado por lo que sus gráficas ya son casi lineales,

$O(n\log(n))$ para ser correctos.

Finalmente y a sorpresa personal tenemos que admitir que Shell Sort es un gran algoritmo y con esto, con este pequeño resultado puedo entender que hay mucho más que la BigO notation cuando nuestro mejor algoritmo fue uno cuadrático, incluso superando a un árbol con información muy bien balanceada.

Hay mucho mas en el análisis de algoritmos que la notación $O()$, mucho más.

0.8.3. Laura

Hablar de algoritmos de ordenamiento siempre me pareció tedioso, cuando se realizó esta practica pude finalmente ver la diferencia entre uno y otro, comienzas a darte cuenta realmente cuanta diferencia puede haber entre un algoritmo con polinomio cuadrático y uno $nlog_n$.

Otra cosa interesante a notar es la diferencia enorme diferencia entre probar los algoritmos en una computadora de 64 bits a una de 32 bits, resulta que para los algoritmos n^2 con n=200,000. Empezaba a tardarse bastante, las pruebas fueron presentadas en una de 64 bits para poder mostrar la ejecución para n mayores a 200,000, pero, si lo ponemos en perspectiva, las computadoras que aun operan a 32 bits o dispositivos que aun no cuentan con los recursos de computadoras de nueva generación suelen tardarse aún más para ordenar, ahora, si usáramos Bubble Sort en todos los dispositivos simplemente por su fácil implementación entonces tendríamos dispositivos muy lentos, o incluso sin ir a dispositivos viejos, los celulares smartphones también requieren realizar este tipo de calculo y cuando le damos un algoritmo ineficiente la rapidez y optimización se ve altamente reducida.

He de decir que quedé impresionada con la cantidad de maneras que hay para ordenar cosas, después me enteré que eso es gran parte de loq ue hace una computadora y entendí el porque la cantidad monstruosa de maneras de realizar este proceso.

Los explicare de la manera más sencilla que se me ocurre:

Bubble Sort: Tomas los 2 primeros y los intercambias si es necesario, tomas los siguientes y los intercambias si es necesario, hasta que recorras tus números hasta el final y no cambies nada.

Bubble Sort v2: Lo mismo que el primero pero, como siempre tendremos el mayor al final podemos evitar recorrer el arreglo completo, así que cada vuelta lo reducimos a n-1

Bubble Sort v3: Lo mismo que el anterior simplemente le preguntamos si ya esta ordenado, si si ya no hace más.

Selection Sort: Escoge el mas pequeño y velo colocando en tu arreglo al principio, el siguiente será después del primero y sucesivamente.

Insertion Sort: Busca el lugar para cada elemento del arreglo mientras lo lees.

Shell Sort: Arreglaos por intervalos, por ejemplo cada 3 números, el primero y el tercero, el tercero y el quinto, y vas bajando tu intervalo.

BTS: Simplemente mete tu información a un árbol con sus reglas básicas y cuando lo recorras de izquierda a derecha lo tendrás ordenado.

La verdad es que explicados así ya no son tan terroríficos como lucían hace un par de años cuando el Bubble no me salia.

Además de aprender esto, pude ver diferente maneras de usar herramientas que realmente pocos profesores te enseñan, como los scripts, aunque usamos python para darle las instrucciones, o simplemente obtener la infracción desde un archivo que puedas definir desde que lo corres, son cosas que funcionan bastante bien pero no nos habíamos dado a la tarea de buscarlas y sobre todo muchas veces ni sabíamos de su existencia.

Me siento engañada por el BTS, pues me habían hablado de el y no pude ver la grandeza de su esplendor, se que es mejor pero supongo que al menos hoy o pude ver toda la capacidad de este algoritmo que no es muy intuitivo pero al parecer muy eficaz.

Anexos

.1. Estructura de directorios

Para que todo funcione correctamente, organizar los archivos y directorios de la siguiente manera:

```
code
├── AuxFunctions.c
├── BubbleSort.c
├── Input10Million.txt .2 InsertionSort.c
├── Make.py
├── SelectionSort.c
├── ShellSort.c
├── SortWithBST.c
├── TestSortAlgorithms.c
├── Time.c
├── Time.h
└── TreeAuxFunction.c
graphics
outputs
```

.2. Código fuente original

.2.1. AuxFunctions.c

```
1  c+cm/*============================================================================
2  c+cm=================    METADATA OF THE FILE      ======================
3  c+cm============================================================================*/
4  c+cm/**
5  c+cm * @author  Rosas Hernandez Oscar Andres
6  c+cm * @author  Alan Enrique Ontiveros Salazar
7  c+cm * @author  Laura Andrea Morales
8  c+cm * @version 0.1
9  c+cm * @team    CompilandoConocimiento
10 c+cm * @date    4/03/2018
11 c+cm */
12
13
14 c+cm/*============================================================================
15 c+cm====================      TEMPORAL SWAP        ======================
16 c+cm============================================================================*/
17 c+cm/**
18 c+cm * Inline function to no create new function stack call to swap
19 c+cm * elements
20 c+cm *
21 c+cm * @param A     A pointer to an int
22 c+cm * @param B     A pointer to an int
```

```
23 c+cm * @return      Nothing...
24 c+cm */
25 kextern k+krinline k+ktvoid n+nfSwapp(k+ktint o*nAp, k+ktint o*nBp) p
26     k+ktint nTemporalp;                                              c+c1//Create
    ↪  a Temporal var
27
28     nTemporal o= o*nAp;                                             c+c1//Tem  A
29     o*nA o= o*nBp;                                                   c+c1//A    B
30     o*nB o= nTemporalp;                                             c+c1//B    Tem
31 p
32
33 c+cm/*=================================================================
34 c+cm====================      PRINT ARRAY          ====================
35 c+cm=================================================================*/
36 c+cm/**
37 c+cm * Inline function to no create new function stack call to print
38 c+cm * the array
39 c+cm *
40 c+cm * @param Data      A pointer to the array of int to sort
41 c+cm * @param DataSize  The size of the Data array
42 c+cm * @param FileName  A pointer to a file to write the array
43 c+cm * @return          Nothing...
44 c+cm */
45 kextern k+krinline k+ktvoid n+nffPrintArrayp(
46     k+ktint nDatap[], k+ktint nDataSizep, k+ktFILE o* nFileNamep) p
    ↪  c+c1//Long parameters
47
48     kfor p(k+ktint ni o= l+m+mi0p; ni o nDataSizep; o++nip)
    ↪  c+c1//Foreach element:
49         nfprintfp(nFileNamep, l+si l+s+senl+sp, nDatap[nip]);
    ↪  c+c1//Print it!
50
51 p
```

.2.2. TreeAuxFunction.c

```
1  c+cm/*=================================================================
2  c+cm==================    METADATA OF THE FILE      ====================
3  c+cm=================================================================*/
4  c+cm/**
5  c+cm * @author  Rosas Hernandez Oscar Andres
6  c+cm * @author  Alan Enrique Ontiveros Salazar
7  c+cm * @author  Laura Andrea Morales
8  c+cm * @version 0.1
9  c+cm * @team    CompilandoConocimiento
10 c+cm * @date    4/03/2018
11 c+cm */
12
13 c+c1// ===================================
14 c+c1// ==== DECLARATION OF A BINARYTREE  ==
```

```
15 c+c1// ===================================
16 ktypedef kstruct nNode p                                          c+c1//===
    ↪  NODE ===
17   k+ktint  nNodeItemp;
    ↪  c+c1//Pointer to the real data
18   kstruct nNode o*nLeftp;
    ↪  c+c1//Pointer to the left node
19   kstruct nNode o*nRightp;
    ↪  c+c1//Pointer to the left node
20 p nNodep;                                                        c+c1//We call
    ↪  this struct a node
21
22 ktypedef nNode nBinaryTreep;                                      c+c1//New
    ↪  name same functionality
23
24
25
26
27 c+cm/*=============================================================
28 c+cm==========        CREATE A BINARY TREE         ===================
29 c+cm=============================================================*/
30 c+cm/**
31 c+cm * Create a Pointer to a Binary Tree that have the data
32 c+cm * that i give you. Note inline
33 c+cm *
34 c+cm * @param NewItem   The data that the node will save
35 c+cm * @return          (Node*) A pointer to the new node
36 c+cm */
37 kextern k+krinline nNodeo* n+nfCreateBinaryTreep(k+ktint nNewItemp) p
    ↪  c+c1// ==== CREATE A NEW NODE ==
38    nNode o*nNewNode o= p(nNodeo*p) nmallocp(ksizeofp(nNodep));
    ↪  c+c1//Reserve memory for  node
39    nNewNodeonNodeItem o= nNewItemp;                               c+c1//You
    ↪  will protect this
40    nNewNodeonLeft o= n+nbNULLp;
    ↪  c+c1//And maybe youre a leaf
41    nNewNodeonRight o= n+nbNULLp;
    ↪  c+c1//And maybe youre a leaf
42    kreturn nNewNodep;                                            c+c1//Go, go
    ↪  NewNode :)
43 p
44
45
46 c+cm/*=============================================================
47 c+cm========          INSERT IN A BINARY SEARCH TREE       ==============
48 c+cm=============================================================*/
49 c+cm/**
50 c+cm * Create a Pointer to a Binary SEARCH Tree that have the data
51 c+cm * that i give you, note that this algorithm is an implementation
52 c+cm *  of a famous algorithm but this is not recursive
53 c+cm *
```

```
54 c+cm * @param Tree      A pointer to a pointer to a Tree struct
55 c+cm * @param NewItem   The data to be inserted
56 c+cm * @return          Nothing...
57 c+cm */
58 k+ktvoid n+nfIterativeInsertBSTp(nBinaryTree o**nTreep, k+ktint nNewItemp) p
    ↪  c+c1// ==== INSERT IN A TREE ==
59    nNode o**nNewNode o= nTreep;                                          c+c1//Let
    ↪  start at root
60
61    kwhile p(o*nNewNode o!= n+nbNULLp)
    ↪  c+c1//While are not at a leaf
62       nNewNode o= p(nNewItem o p(o*nNewNodep)onNodeItemp)o?
    ↪  c+c1//We have to move right
63            op((o*nNewNodep)onLeftp)o: op((o*nNewNodep)onRightp);
    ↪  c+c1//Move left or right
64
65    p(o*nNewNodep) o= nCreateBinaryTreep(nNewItemp);
    ↪  c+c1//Create a node at a leaf
66 p
67
68 c+cm/*================================================================
69 c+cm========   IN ORDER TRANSVERSE OF A BINARY SEARCH TREE      =========
70 c+cm================================================================*/
71 c+cm/**
72 c+cm * This will give you an array fill with the data of a tree
73 c+cm * if you transverse it inorder
74 c+cm *
75 c+cm * @param Tree      A pointer to a pointer to a Tree struct
76 c+cm * @param Data[]    This is a pointer to an ALREADY reserve Data
77 c+cm                    size dimension
78 c+cm * @return          Nothing...
79 c+cm */
80 k+ktvoid n+nfIterativeCreateInOrderp(nNode o**nTreep, k+ktint o*nDatap, k+ktint
    ↪  nDataSizep) p c+c1//= CREATE ARRAY FROM TREE ==
81    nNode o*nActualNode o= o*nTreep;
    ↪  c+c1//Now, use a temporal node
82
83    nNode o**nStack o= ncallocp(nDataSizep, ksizeofp(nNodeo*p));
    ↪  c+c1//Create a stack like
84
85    k+ktint nStackPointer o= ol+m+mi1p, ni o= l+m+mi0p;
    ↪  c+c1//Aux variables
86
87    kdo p                                                          c+c1//Do the
    ↪  next:
88
89       kwhile p(nActualNode o!= n+nbNULLp) p
    ↪  c+c1//While we are not a leaf
90            nStackp[o++nStackPointerp] o= nActualNodep;
    ↪  c+c1//Add to stack the node
```

```
91              nActualNode o= nActualNodeonLeftp;                              c+c1//Move
    ↪  to the fuck to left
92              p
93
94              kif p(nStackPointer o= l+m+mi0p) p
    ↪  c+c1//Ok, now while not empty
95              nActualNode o= nStackp[nStackPointerop];
    ↪  c+c1//Start pushing the data
96              nDatap[nio++p] o= nActualNodeonNodeItemp;
    ↪  c+c1//Now, add to the data array
97              nActualNode o= nActualNodeonRightp;                             c+c1//And
    ↪  move to right
98              p
99          p
100         kwhile p(nActualNode o!= n+nbNULL o|| nStackPointer o= l+m+mi0p);
    ↪  c+c1//Now do this while we can
101
102         nfreep(nStackp);                                                    c+c1//Bye
    ↪  Stack :)
103 p
```

.2.3. Time.h

```
1  c+c1//*****************************************************************
2  c+c1//TIEMPO.H
3  c+c1//*****************************************************************
4  c+c1//*****************************************************************
5  c+c1//M. EN C. EDGARDO ADRIÁN FRANCO MARTÍNEZ
6  c+c1//Curso: Análisis de algoritmos
7  c+c1//(C) Enero 2013
8  c+c1//ESCOMIPN
9  c+c1//Ejemplo de medición de tiempo en C y recepción de parametros en C bajo UNIX
10 c+c1//Compilación de la libreria: gcc c tiempo.c  (Generación del código objeto)
11 c+c1//*****************************************************************
12
13
14 c+c1//*****************************************************************
15 c+c1//uswtime (Declaración)
16 c+c1//*****************************************************************
17 c+c1//Descripción: Función que almacena en las variables referenciadas
18 c+c1//el tiempo de CPU, de E/S y Total actual del proceso actual.
19 c+c1//
20 c+c1//Recibe: Variables de tipo doble para almacenar los tiempos actuales
21 c+c1//Devuelve:
22 c+c1//*****************************************************************
23 k+ktvoid n+nfuswtimep(k+ktdouble o*nusertimep, k+ktdouble o*nsystimep, k+ktdouble
    ↪  o*nwalltimep);
```

.2.4. Time.c

```
c+c1//*****************************************************************
c+c1//TIEMPO.C
c+c1//*****************************************************************
c+c1//*****************************************************************
c+c1//M. EN C. EDGARDO ADRIÁN FRANCO MARTÍNEZ
c+c1//Curso: Análisis de algoritmos
c+c1//(C) Enero 2013
c+c1//ESCOMIPN
c+c1//Ejemplo de medición de tiempo en C y recepción de parametros en C bajo UNIX
c+c1//Compilación de la libreria: gcc c tiempo.c  (Generación del código objeto)
c+c1//*****************************************************************


c+c1//*****************************************************************
c+c1//Librerias incluidas
c+c1//*****************************************************************
c+cpinclude c+cpfsys/resource.h
c+cpinclude c+cpfsys/time.h
c+cpinclude c+cpfTime.h

c+c1//*****************************************************************
c+c1//uswtime (Definición)
c+c1//*****************************************************************
c+c1//Descripción: Función que almacena en las variables referenciadas
c+c1//el tiempo de CPU, de E/S y Total actual del proceso actual.
c+c1//
c+c1//Recibe: Variables de tipo doble para almacenar los tiempos actuales
c+c1//Devuelve:
c+c1//*****************************************************************include
 ↪  stdio.h
k+ktvoid n+nfuswtimep(k+ktdouble o*nusertimep, k+ktdouble o*nsystimep, k+ktdouble
 ↪  o*nwalltimep)
p
    k+ktdouble nmega o= l+m+mf1.0e6p;
    kstruct nrusage nbufferp;
    kstruct ntimeval ntpp;
    ngetrusagep(nRUSAGESELFp, onbufferp);
    ngettimeofdayp(ontpp, l+m+mi0p);
    o*nusertime o= p(k+ktdoublep) nbufferp.nruutimep.ntvsec o+l+m+mf1.0e6 o*
 ↪  nbufferp.nruutimep.ntvusecp;
    o*nsystime  o= p(k+ktdoublep) nbufferp.nrustimep.ntvsec o+l+m+mf1.0e6 o*
 ↪  nbufferp.nrustimep.ntvusecp;
    o*nwalltime o= p(k+ktdoublep) ntpp.ntvsec o+ l+m+mf1.0e6 o* ntpp.ntvusecp;
p

c+cm/*En Unix, se dispone de temporizadores ejecutables (en concreto time) que nos
 ↪  proporcionan medidas de los tiempos
c+cmde ejecución de programas. Estos temporizadores nos proporcionan tres medidas
 ↪  de tiempo:
```

```
44
45 c+cm    * real:  Tiempo real que se ha tardado desde que se lanzó el programa a
    ↪ ejecutarse hasta que el programa finalizó y proporcionó los resultados.
46 c+cm    * user: Tiempo que la CPU se ha dedicado exclusivamente a la computación
    ↪ del programa.
47 c+cm    * sys:     Tiempo que la CPU se ha dedicado a dar servicio al sistema
    ↪ operativo por necesidades del programa (por ejemplo para llamadas al sistema
    ↪ para efectuar I/O).
48
49 c+cmEl tiempo real también suele recibir el nombre de elapsed time o wall time.
    ↪ Algunos temporizadores también proporcionan el porcentaje de tiempo que la CPU
    ↪ se ha dedicado al programa. Este porcentaje viene dado por la relación entre el
    ↪ tiempo de CPU (user + sys)
50 c+cmy el tiempo real, y da una idea de lo cargado que se hallaba el sistema en el
    ↪ momento de la ejecución del programa.
51
52 c+cmEl grave inconveniente de los temporizadores ejecutables es que no son capaces
    ↪ de proporcionar medidas de tiempo de ejecución de segmentos de código. Para
    ↪ ello, hemos de invocar en nuestros propios programas  a un conjunto de
    ↪ temporizadores disponibles en la mayor parte de las librerías de C de Unix, que
    ↪ serán los que nos proporcionen medidas sobre los tiempos de ejecución de trozos
    ↪ discretos de código.
53
54 c+cmEn nuestras prácticas vamos a emplear una función que actúe de temporizador y
    ↪ que nos proporcione los tiempos de CPU (user, sys)
55 c+cmy el tiempo real. En concreto, vamos a emplear el procedimiento uswtime listado
    ↪ a continuación.
56
57 c+cmEste procedimiento en realidad invoca a dos funciones de Unix: getrusage y
    ↪ gettimeofday. La primera de ellas nos proporciona el tiempo de CPU, tanto de
    ↪ usuario como de sistema, mientras que la segunda nos proporciona el tiempo real
    ↪ (wall time). Estas dos funciones son las que disponen de mayor resolución de
    ↪ todos los temporizadores disponibles en Unix.
58 c+cm
59 c+cmModo de Empleo:
60 c+cm
61 c+cmLa función uswtime se puede emplear para medir los tiempos de ejecución de
    ↪ determinados segmentos de código en nuestros programas. De forma esquemática,
    ↪ el empleo de esta función constaría de los siguientes pasos:
62
63 c+cm    1. Invocar a uswtime para fijar el instante a partir del cual se va a medir
    ↪ el tiempo.
64
65 c+cm            uswtime(utime0, stime0, wtime0);
66
67 c+cm    2. Ejecutar el código cuyo tiempo de ejecución se desea medir.
68 c+cm    3. Invocar a uswtime para establecer el instante en el cual finaliza la
    ↪ medición
69 c+cm        del tiempo de ejecución.
70
71 c+cm            uswtime(utime1, stime1, wtime1);
```

```
72
73 c+cm    4. Calcular los tiempos de ejecución como la diferencia entre la primera y
    ↪  segunda
74 c+cm        invocación a uswtime:
75
76 c+cm            real:   wtime1  wtime0
77 c+cm            user:   utime1  utime0
78 c+cm            sys :   stime1  stime0
79
80 c+cm            El porcentaje de tiempo dedicado a la ejecución de ese segmento de
    ↪  código
81 c+cm            vendría dado por la relación CPU/Wall:
82
83 c+cm    CPU/Wall = (user + sys) / real x 100 */
```

.2.5. BubbleSort.c

```
1  c+cm/*==========================================================================
2  c+cm==================    METADATA OF THE FILE      ====================
3  c+cm==========================================================================*/
4  c+cm/**
5  c+cm * @author  Rosas Hernandez Oscar Andres
6  c+cm * @author  Alan Enrique Ontiveros Salazar
7  c+cm * @author  Laura Andrea Morales
8  c+cm * @version 0.1
9  c+cm * @team    CompilandoConocimiento
10 c+cm * @date    4/03/2018
11 c+cm */
12
13
14 c+cm/*==========================================================================
15 c+cm====================      BUBBLE SORT          ====================
16 c+cm==========================================================================*/
17
18 c+cm/**
19 c+cm * This is the most stupid (not counting stupid sort) no optimization
20 c+cm * nothing, the raw algorithm.
21 c+cm *
22 c+cm * For each element in the Array check if there are 2 contiguos
23 c+cm * elements that are in the wrong order, to swap it!
24 c+cm *
25 c+cm * @param Data      A pointer to the array of int to sort
26 c+cm * @param DataSize  The size of the Data array
27 c+cm * @return          Nothing...Im modifying the raw data
28 c+cm */
29 k+ktvoid n+nfBubbleSortv1p(k+ktint nDatap[], k+ktint nDataSizep) p
    ↪  c+c1//=== BUBBLE SORT ======
30     kfor p(k+ktint ni o= l+m+mi0p; ni o nDataSize o l+m+mi1p; o++nip) p
    ↪  c+c1//Do this Size  1 times
```

```
31          kfor p(k+ktint nj o= l+m+mi0p; nj o nDataSize o l+m+mi1p; o++njp) p
 ↪  c+c1//Do this Size  1 times
32              kif p(nDatap[njp] o nDatap[nj o+ l+m+mi1p])
 ↪  c+c1//If we need to swap it
33                  nSwapp(onDatap[njp], onDatap[nj o+ l+m+mi1p]);
 ↪  c+c1//Swap it!
34          p
35      p
36  p
37
38
39
40  c+cm/*=============================================================================
41  c+cm=====================      BUBBLE SORT          =====================
42  c+cm=============================================================================*/
43  c+cm/**
44  c+cm * This is the most important optimization, reduce the second
45  c+cm * loop because last i elements are already in place
46  c+cm *
47  c+cm * @see BubbleSortv1
48  c+cm */
49  k+ktvoid n+nfBubbleSortv2p(k+ktint nDatap[], k+ktint nDataSizep) p
50      kfor p(k+ktint ni o= l+m+mi0p; ni o nDataSize o l+m+mi1p; nio++p) p
 ↪  c+c1//Do this Size  1 times
51          kfor p(k+ktint nj o= l+m+mi0p; nj o nDataSize o ni o l+m+mi1p; njo++p) p
 ↪  c+c1//Last i are sorted
52              kif p(nDatap[njp] o nDatap[nj o+ l+m+mi1p])
 ↪  c+c1//If we need to swap it
53                  nSwapp(onDatap[njp], onDatap[nj o+ l+m+mi1p]);
 ↪  c+c1//Swap it!
54          p
55      p
56  p
57
58
59  c+cm/*=============================================================================
60  c+cm=====================      BUBBLE SORT          =====================
61  c+cm=============================================================================*/
62  c+cm/**
63  c+cm * This is the next most important optimization, break the seach
64  c+cm * if is already sorted
65  c+cm *
66  c+cm * @see BubbleSortv1
67  c+cm */
68  k+ktvoid n+nfBubbleSortv3p(k+ktint nDatap[], k+ktint nDataSizep) p
69
70      kfor p(k+ktint ni o= l+m+mi0p; ni o nDataSize o l+m+mi1p; nio++p) p
 ↪  c+c1//Do: Size  1 times
71          k+ktbool nSwapped o= n+nbfalsep;
 ↪  c+c1//Suposse no swap
72
```

```
73          kfor p(k+ktint nj o= l+m+mi0p; nj o nDataSize o ni o l+m+mi1p; njo++p) p
    ↪  c+c1//Last i are sorted
74              kif p(nDatap[njp] o nDatap[nj o+ l+m+mi1p]) p
    ↪  c+c1//We need to swap it?
75                  nSwapp(onDatap[njp], onDatap[nj o+ l+m+mi1p]);
    ↪  c+c1//Swap it!
76                  nSwapped o= n+nbtruep;
    ↪  c+c1//Upps! swap
77              p
78          p
79
80          kif p(o!nSwappedp) kbreakp;                                    c+c1//No
    ↪  swap,Its sorted!
81      p
82  p
```

.2.6. InsertionSort.c

```
1   c+cm/*==========================================================================
2   c+cm==================    METADATA OF THE FILE      =====================
3   c+cm==========================================================================*/
4   c+cm/**
5   c+cm * @author  Rosas Hernandez Oscar Andres
6   c+cm * @author  Alan Enrique Ontiveros Salazar
7   c+cm * @author  Laura Andrea Morales
8   c+cm * @version 0.1
9   c+cm * @team    CompilandoConocimiento
10  c+cm * @date    4/03/2018
11  c+cm */
12
13
14  c+cm/*==========================================================================
15  c+cm==================      INSERTION SORT          =====================
16  c+cm==========================================================================*/
17
18  c+cm/**
19  c+cm * This is a really intuitive sorting algorithm, to do so we say
20  c+cm * that we will create a new subarray.
21  c+cm * So the subarray [0, 1] is already sorted, now:
22  c+cm * Take the next element contiguos to the subarray an put it where it
23  c+cm * belongs and move all the other 1 place.
24  c+cm * Now my new subarray is from [0, 2], and repeat, you get the point
25  c+cm *
26  c+cm * @param Data      A pointer to the array of int to sort
27  c+cm * @param DataSize  The size of the Data array
28  c+cm * @return          Nothing...Im modifying the raw data
29  c+cm */
30  k+ktvoid n+nfInsertionSortp(k+ktint nDatap[], k+ktint nDataSizep)
    ↪  c+c1//= INSERTION SORT ===
31
```

```
32      kfor p(k+ktint ni o= l+m+mi1p; ni o nDataSizep; o++nip) p
    ↪   c+c1//Traverse each item
33
34          k+ktint nj o= nip;
    ↪   c+c1//A[0..i1] is already sorted
35          k+ktint nTemp o= nDatap[nip];
    ↪   c+c1//Lets find next item
36
37          kwhile p(nj o l+m+mi0 o nTemp o nDatap[nj o l+m+mi1p]) p
    ↪   c+c1//Move elements of the subarray
38              nDatap[njp] o= nDatap[nj o l+m+mi1p];
    ↪   c+c1//to one element ahead
39              njop;                                                   c+c1//until we
    ↪   find the correct place
40          p
41
42          nDatap[njp] o= nTempp;
    ↪   c+c1//Put there the Temp
43      p
44  p
```

.2.7. SelectionSort.c

```
1   c+cm/*============================================================================
2   c+cm=================    METADATA OF THE FILE      ====================
3   c+cm============================================================================*/
4   c+cm/**
5   c+cm * @author  Rosas Hernandez Oscar Andres
6   c+cm * @author  Alan Enrique Ontiveros Salazar
7   c+cm * @author  Laura Andrea Morales
8   c+cm * @version 0.1
9   c+cm * @team    CompilandoConocimiento
10  c+cm * @date    4/03/2018
11  c+cm */
12
13
14  c+cm/*============================================================================
15  c+cm=================      SELECTION SORT          ====================
16  c+cm============================================================================*/
17
18  c+cm/**
19  c+cm * Select the lowest value for each index
20  c+cm * Ok, so it in escence we find the smallest data, then we put it
21  c+cm * at the begging, next me only have to sort the array starting
22  c+cm * in the next position, so we do the same thing another time
23  c+cm *
24  c+cm * @param Data      A pointer to the array of int to sort
25  c+cm * @param DataSize  The size of the Data array
26  c+cm * @return          Nothing...Im modifying the raw data
27  c+cm */
```

```
28
29 k+ktvoid n+nfSelectionSortp(k+ktint nDatap[], k+ktint nDataSizep)
    ↪  c+c1//== SELECTION SORT =
30
31     kfor p(k+ktint ni o= l+m+mi0p; ni o nDataSize o l+m+mi1p; o++nip) p
    ↪  c+c1//For each index
32         k+ktint nTiniestIndex o= nip;
    ↪  c+c1//Save actual index
33         kfor p(k+ktint nj o= ni o+ l+m+mi1p; nj o nDataSizep; o++njp)
    ↪  c+c1//Check if is tinest
34             kif p(nDatap[njp] o nDatap[nTiniestIndexp])
    ↪  c+c1//If exists smallest
35                 nTiniestIndex o= njp;
    ↪  c+c1//Change the index
36
37         nSwapp(onDatap[nTiniestIndexp], onDatap[nip]);
    ↪  c+c1//Swap tiniest data
38     p
39 p
```

.2.8. ShellSort.c

```
1  c+cm/*============================================================================
2  c+cm==================    METADATA OF THE FILE      =====================
3  c+cm============================================================================*/
4  c+cm/**
5  c+cm * @author  Rosas Hernandez Oscar Andres
6  c+cm * @author  Alan Enrique Ontiveros Salazar
7  c+cm * @author  Laura Andrea Morales
8  c+cm * @version 0.1
9  c+cm * @team    CompilandoConocimiento
10 c+cm * @date    4/03/2018
11 c+cm */
12
13
14 c+cm/*============================================================================
15 c+cm=====================      SHELL SORT           =====================
16 c+cm============================================================================*/
17
18 c+cm/**
19 c+cm * ShellSort is mainly a variation of Insertion Sort.
20 c+cm * In insertion sort, we move elements only one position ahead.
21 c+cm * When an element has to be moved far ahead, many movements are involved.
22 c+cm * The idea of shellSort is to allow exchange of far items. In shellSort, we
23 c+cm * make the array hsorted for a large value of h. We keep reducing the value
24 c+cm * of h until it becomes 1.
25 c+cm *
26 c+cm * An array is said to be hsorted if all sublists of every h'th element is
    ↪  sorted.
27 c+cm *
```

```
28 c+cm * @param Data      A pointer to the array of int to sort
29 c+cm * @param DataSize  The size of the Data array
30 c+cm * @return          Nothing...Im modifying the raw data
31 c+cm */
32
33 k+ktvoid n+nfShellSortp(k+ktint nDatap[], k+ktint nDataSizep) p
    ↪  c+c1//=== SHELL SORT ======
34     k+ktint nGap o= nDataSize o l+m+mi1p;
    ↪  c+c1//Let jump = DataSize / 2
35
36     kwhile p(nGap o l+m+mi0p) p
    ↪  c+c1//Until entire data sort
37
38         kforp(k+ktint ni o= nGapp; ni o nDataSizep; nio++p) p
    ↪  c+c1//For each subarray
39
40             k+ktint nj o= nip;
    ↪  c+c1//Let j = i to modify j
41             k+ktint nTemporal o= nDatap[nip];
    ↪  c+c1//Temp. will be the next
42
43             kwhile p(nj o= nGap o nTemporal o nDatap[nj o nGapp]) p
    ↪  c+c1//shift earlier gapsort
44                 nDatap[njp] o= nDatap[nj o nGapp];
    ↪  c+c1//until correct place
45                 nj o= nGapp;                                          c+c1//is
    ↪  found for Data[i]
46             p
47
48             nDatap[njp] o= nTemporalp;
    ↪  c+c1//Temp. find their place
49         p
50
51         nGap o= l+m+mi1p;
    ↪  c+c1//Reduce gap in half
52     p
53 p
```

.2.9. SortWithBST.c

```
1  c+cm/*================================================================
2  c+cm==================    METADATA OF THE FILE      ====================
3  c+cm================================================================*/
4  c+cm/**
5  c+cm * @author  Rosas Hernandez Oscar Andres
6  c+cm * @author  Alan Enrique Ontiveros Salazar
7  c+cm * @author  Laura Andrea Morales
8  c+cm * @version 0.1
9  c+cm * @team    CompilandoConocimiento
10 c+cm * @date    4/03/2018
```

```
c+cm */

c+cpinclude c+cpfTreeAuxFunction.c
c+cm/*================================================================
c+cm=================           BST SORT           ====================
c+cm================================================================*/

c+cm/**
c+cm * Ok, i dont have much to say, we will put it in a Binary Search Tree
c+cm * ant the transverse it, thats it!
c+cm *
c+cm * @param Data      A pointer to the array of int to sort
c+cm * @param DataSize  The size of the Data array
c+cm * @return          Nothing...Im modifying the raw data
c+cm */
k+ktvoid n+nfSortWithBSTp(k+ktint nDatap[], k+ktint nDataSizep) p
    c+c1//=== BST SORT ======
   nNodeo* nTree o= n+nbNULLp;
    c+c1//Start with empty tree

   kforp(k+ktint ni o= l+m+mi0p; ni o nDataSizep; nio++p)
    c+c1//For each element
      nIterativeInsertBSTp(onTreep, nDatap[nip]);
    c+c1//Insert in tree

   nIterativeCreateInOrderp(onTreep, nDatap, nDataSizep);
    c+c1//Transverse it!
p
```

.2.10. TestSortAlgorithms.c

```
c+cm/*================================================================
c+cm=================    METADATA OF THE FILE      ====================
c+cm================================================================*/
c+cm/**
c+cm * @author  Rosas Hernandez Oscar Andres
c+cm * @author  Alan Enrique Ontiveros Salazar
c+cm * @author  Laura Andrea Morales
c+cm * @version 0.1
c+cm * @team    CompilandoConocimiento
c+cm * @date    4/03/2018
c+cm * @compile gcc std=c11 Time.c TestSortAlgorithms.c o TestSortAlgorithms
c+cm * @run ./TestSortAlgorithms n NumAlgorithm OutputPlace  Input10Million.txt
c+cm */


c+cpinclude c+cpfstdio.h
c+cpinclude c+cpfstdlib.h
c+cpinclude c+cpfstdbool.h
c+cpinclude c+cpfstring.h
```

```
20
21  c+cpinclude c+cpfTime.h
22
23  c+cpinclude c+cpfAuxFunctions.c
24  c+cpinclude c+cpfBubbleSort.c
25  c+cpinclude c+cpfSelectionSort.c
26  c+cpinclude c+cpfInsertionSort.c
27  c+cpinclude c+cpfShellSort.c
28  c+cpinclude c+cpfSortWithBST.c
29
30  c+c1// ================================================
31  c+c1// ============     MAIN        =================
32  c+c1// ================================================
33  c+cm/**
34  c+cm * This is the control program to check all the other
35  c+cm * sort algorithms.
36  c+cm *
37  c+cm * @param argc      Size of elements of argv
38  c+cm * @param argv[0]   Name of the program
39  c+cm * @param argv[1]   Size of the array to get
40  c+cm * @param argv[2]   Number of the algorithm
41  c+cm * @return          (Int)A zero if all OK
42  c+cm */
43
44  k+ktint n+nfmainp(k+ktint nargcp, k+ktchar kconst o*nargvp[]) p
45
46      c+c1// === READ AND GET THE DATA  ====
47      kif p(nargc o l+m+mi3p) nexitp(l+m+mi0p);
    ↪   c+c1//Simple check
48
49      k+ktint nDataSize o= natoip(nargvp[l+m+mi1p]);                      c+c1//Get
    ↪   the DataSize
50      k+ktint nAlgorithm o= natoip(nargvp[l+m+mi2p]);                     c+c1//Get
    ↪   the Algorithm
51
52      k+ktdouble nUserTimeStartp, nSysTimeStartp, nWallTimeStartp;  c+c1//Start
    ↪   variables
53      k+ktdouble nUserTimeEndp, nSysTimeEndp, nWallTimeEndp;        c+c1//End
    ↪   variables
54
55      k+ktFILE o* nFileName o= nfopen p(nargvp[l+m+mi3p], l+swp);
    ↪   c+c1//Open the file
56
57      k+ktint o*nOriginalData o=
58              p(k+ktinto*p) nmallocp(nDataSizeo*ksizeofp(k+ktintp));
    ↪   c+c1//Reserve data
59
60      kfor p(k+ktint ni o= l+m+mi0p; ni o nDataSizep; o++nip)
    ↪   c+c1//For each number
61          nscanfp(l+sip, onOriginalDatap[nip]);                 c+c1//Get the number
62
```

```
63      c+c1// === NOW SORT THE DATA ========
64      nuswtimep(onUserTimeStartp,
65              onSysTimeStartp,
66              onWallTimeStartp);                              c+c1//START COUNTING
67
68      kif p(nAlgorithm o== l+m+mi0p)                                  c+c1//0 is
    ↪   Bubble Sort
69          nBubbleSortv1p(nOriginalDatap, nDataSizep);          c+c1//Bubble Sort
70      kelse kif p(nAlgorithm o== l+m+mi1p)                            c+c1//1 is
    ↪   Bubble Sort v2
71          nBubbleSortv2p(nOriginalDatap, nDataSizep);          c+c1//Bubble Sort v2
72      kelse kif p(nAlgorithm o== l+m+mi2p)                            c+c1//2 is
    ↪   Bubble Sort v3
73          nBubbleSortv3p(nOriginalDatap, nDataSizep);          c+c1//Bubble Sort v3
74      kelse kif p(nAlgorithm o== l+m+mi3p)                            c+c1//3 is
    ↪   SelectionSort
75          nSelectionSortp(nOriginalDatap, nDataSizep);         c+c1//SelectionSort
76      kelse kif p(nAlgorithm o== l+m+mi4p)                            c+c1//4 is
    ↪   InsertionSort
77          nInsertionSortp(nOriginalDatap, nDataSizep);         c+c1//InsertionSort
78      kelse kif p(nAlgorithm o== l+m+mi5p)                            c+c1//5 is
    ↪   ShellSort
79          nShellSortp(nOriginalDatap, nDataSizep);             c+c1//ShellSort
80      kelse kif p(nAlgorithm o== l+m+mi6p)                            c+c1//6 is
    ↪   SortWithBST
81          nSortWithBSTp(nOriginalDatap, nDataSizep);           c+c1//SortWithBST
82
83      nuswtimep(
84          onUserTimeEndp,
85          onSysTimeEndp,
86          onWallTimeEndp);                                    c+c1//END THE COUNT
87
88      c+c1// === NOW SHOW THE DATA ========
89      k+ktdouble nRealTime o= nWallTimeEnd o nWallTimeStartp;      c+c1//Get
    ↪   difference
90      k+ktdouble nUserTime o= nUserTimeEnd o nUserTimeStartp;      c+c1//Get
    ↪   difference
91      k+ktdouble nSysTime  o= nSysTimeEnd o nSysTimeStartp;        c+c1//Get
    ↪   difference
92
93      nprintfp(l+s.10f .10f .10fp, nRealTimep, nUserTimep, nSysTimep);
94
95      c+c1// === NOW CREATE THE DATA ========
96      nfPrintArrayp(nOriginalDatap, nDataSizep, nFileNamep);      c+c1//Now sorted
97      nfclosep(nFileNamep);                                     c+c1//Close the name
98
99      kreturn l+m+mi0p;                                            c+c1//Bye friends
100 p
```

.2.11. Make.py

```
k+knimport n+nnos
k+knimport n+nnsubprocess
k+knimport n+nnnumpy k+knas n+nnnp
k+knimport n+nnmatplotlib.pyplot k+knas n+nnplt

l+s+sd/*================================================================
l+s+sd==================    METADATA OF THE FILE      ====================
l+s+sd================================================================*/
l+s+sd/**
l+s+sd * @author  Rosas Hernandez Oscar Andres
l+s+sd * @author  Alan Enrique Ontiveros Salazar
l+s+sd * @author  Laura Andrea Morales
l+s+sd * @version 0.1
l+s+sd * @team    CompilandoConocimiento
l+s+sd * @date    4/03/2018
l+s+sd * @run     python3.6 Make.py
l+s+sd * @require numpy, matplotlib
l+s+sd */
l+s+sd

nDataSize o= p[
    l+m+mi100p, l+m+mi1000p, l+m+mi5000p, l+m+mi10000p, l+m+mi50000p,
 ↪  l+m+mi100000p,
    l+m+mi200000p, l+m+mi400000p, l+m+mi600000p, l+m+mi800000p, l+m+mi1000000p,
 ↪  l+m+mi2000000p,
    l+m+mi3000000p, l+m+mi4000000p, l+m+mi5000000p, l+m+mi6000000p, l+m+mi7000000p,
    l+m+mi8000000p, l+m+mi9000000p, l+m+mi10000000
p]

nDataExtra o= p[l+m+mi50000000p, l+m+mi100000000p, l+m+mi500000000p,
l+m+mi1000000000p, l+m+mi5000000000p]

nAlgorithms o= p[
    l+s+s2BubbleSortv1p,
    l+s+s2BubbleSortv2p,
    l+s+s2BubbleSortv3p,
    l+s+s2SelectionSortp,
    l+s+s2InsertionSortp,
    l+s+s2ShellSortp,
    l+s+s2SortWithBST
p]

nDegrees o= p[l+m+mi1p, l+m+mi2p, l+m+mi3p, l+m+mi4p, l+m+mi8p]

nProgramName o= l+s+s2TestSortAlgorithms
nInput       o= l+s+s2Input10Million.txt
nFlags       o= l+s+s2std=c11 Time.c

noso.nsystemp(l+s+s2resetp)
```

```
48 noso.nsystemp(nFl+s+s2gcc Flags ProgramName.c o ProgramNamep)
49
50 kdef n+nfgraficarp(ndataxp, ndatayp, nlegendsp, nlabelxp, nlabelyp, ntitlep,
    ↪  n+nbfilep, nmarkerp):
51     kfor ni o+owin n+nbrangep(l+m+mi0p, n+nblenp(ndataxp)):
52         nplto.nplotp(ndataxp[nip], ndatayp[nip], nlabel o= nlegendsp[nip], nmarker
    ↪  o= nmarkerp)
53     nplto.ngridp(n+nb+bpTruep)
54     nplto.nxlabelp(nlabelxp)
55     nplto.nylabelp(nlabelyp)
56     nplto.ntitlep(ntitlep)
57     nplto.nlegendp(nloco=l+s+s1upper centerp, nbboxtoanchoro=p(l+m+mf0.5p,
    ↪  ol+m+mf0.1p), nshadowo=n+nb+bpTruep, nncolo=l+m+mi4p)
58     nplto.nsavefigp(n+nbfilep, nbboxincheso=l+s+s1tightp)
59     nplto.nclfp()
60
61 nxrealall o= p[]
62 nyrealall o= p[]
63 nxpolyall o= p[]
64 nypolyall o= p[]
65 npolyall   o= p[]
66
67 n+nbfile o= n+nbopenp(l+s+s2../outputs/info.txtp, l+s+s2wp)
68
69 kfor nNumAlgorithm o+owin n+nbrangep(l+m+mi0p, n+nblenp(nAlgorithmsp)):
70
71     nAlgorithmName o= nAlgorithmsp[nNumAlgorithmp]
72
73     nx o= p[];
74     nyreal o= p[];
75     nycpu o= p[];
76     nyes o= p[];
77
78     kfor nn o+owin nDataSizep:
79
80         kif nNumAlgorithm o= l+m+mi5 o+owor nn o= l+m+mi1000000p:
81
82             nOutputPlace o= nfl+s+s2../outputs/OutAlgorithmNameN=n
83
84             kprintp(nfl+s+s2Running Algorithm AlgorithmName for n = np)
85
86             nOutputProgram o= nsubprocesso.ncheckoutputp(
87                 nfl+s+s1./ProgramName n NumAlgorithm OutputPlace  Inputp,
88                 nshell o= n+nb+bpTruep,
89                 nuniversalnewlines o= n+nb+bpTruep)
90
91             nData o= p[n+nbfloatp(nip) kfor ni o+owin nOutputProgramo.nsplitp()]
92             nRealTime o= nDatap[l+m+mi0p]
93             nUserTime o= nDatap[l+m+mi1p]
94             nSysTime o= nDatap[l+m+mi2p]
95             nCPUWall o= p(nUserTime o+ nSysTimep) o/ nRealTimep;
```

```
96
97              nxo.nappendp(nnp)
98              nyrealo.nappendp(nRealTimep)
99              nycpuo.nappendp(nUserTimep)
100             nyeso.nappendp(nSysTimep)
101
102             kprintp(nfl+s+s2Real:     RealTimesp)
103             kprintp(nfl+s+s2User:     UserTimesp)
104             kprintp(nfl+s+s2Sys:      SysTimesp)
105             kprintp(nfl+s+s2CPU/Wall: CPUWall * 100p)
106             kprintp(l+s+s2p)
107
108     ngraficarp([nxp, nxp, nxp], p[nyrealp, nycpup, nyesp], p[l+s+s2Tiempo realp,
    ↪  l+s+s2Tiempo CPUp, l+s+s2Tiempo E/Sp],
109         l+s+s2Tamaño del problema (n)p, l+s+s2Tiempo (s)p, nAlgorithmNamep,
110         nfl+s+s2../graphics/AlgorithmNameExperimentalTimes.pngp, l+s+s1op)
111
112     n+nbfileo.nwritep(l+s+s2=== o+ nAlgorithmName o+ l+s+s2===l+s+senl+s+s2p)
113     n+nbfileo.nwritep(l+s+s2 Real times:l+s+senl+s+s2p)
114     kfor ni o+owin n+nbrangep(l+m+mi0p, n+nblenp(nxp)):
115         n+nbfileo.nwritep(nfl+s+s2(x[i], yreal[i])l+s+senl+s+s2p)
116     n+nbfileo.nwritep(l+s+s2 User times:l+s+senl+s+s2p)
117     kfor ni o+owin n+nbrangep(l+m+mi0p, n+nblenp(nxp)):
118         n+nbfileo.nwritep(nfl+s+s2(x[i], ycpu[i])l+s+senl+s+s2p)
119     n+nbfileo.nwritep(l+s+s2 Sys times:l+s+senl+s+s2p)
120     kfor ni o+owin n+nbrangep(l+m+mi0p, n+nblenp(nxp)):
121         n+nbfileo.nwritep(nfl+s+s2(x[i], yes[i])l+s+senl+s+s2p)
122     n+nbfileo.nwritep(l+s+s2l+s+senl+s+s2p)
123
124     nxrealallo.nappendp(nxp)
125     nyrealallo.nappendp(nyrealp)
126
127     npolynomialsx o= p[]
128     npolynomialsy o= p[]
129     kfor ndegree o+owin nDegreesp:
130         nxp o= nnpo.nlinspacep(l+m+mi0p, nxp[ol+m+mi1p], l+m+mi1000p)
131         npolynomialsxo.nappendp(nxpp)
132         npolynomial o= nnpo.npoly1dp(nnpo.npolyfitp(nxp, nyrealp, ndegreep))
133         nevaluations o= npolynomialp(nxpp)
134         npolynomialsyo.nappendp(nevaluationsp)
135         kif p(nNumAlgorithm o= l+m+mi4 o+owand ndegree o== l+m+mi2p) o+owor
    ↪  p(nNumAlgorithm o= l+m+mi5 o+owand ndegree o== l+m+mi1p):
136             nxpolyallo.nappendp(nxpp)
137             nypolyallo.nappendp(nevaluationsp)
138             npolyallo.nappendp(npolynomialp)
139     nplto.nplotp(nxp, nyrealp, l+s+s1op)
140     ngraficarp(npolynomialsxp, npolynomialsyp, p[nfl+s+s2Polinomio grado Degrees[i]
    ↪  kfor ni o+owin n+nbrangep(l+m+mi0p, n+nblenp(nDegreesp))],
141         l+s+s2Tamaño del problema (n)p, l+s+s2Tiempo (s)p, nAlgorithmNamep,
142         nfl+s+s2../graphics/AlgorithmNamePolynomials.pngp, l+s+s1p)
143     kprintp(nend o= l+s+s2l+s+sennl+s+s2p)
```

```
144
145 n+nbfileo.nclosep()
146
147 ninfopoly o= n+nbopenp(l+s+s2../outputs/polynomials.txtp, l+s+s2wp)
148 kfor ni o+owin n+nbrangep(l+m+mi0p, n+nblenp(nAlgorithmsp)):
149     ninfopolyo.nwritep(nAlgorithmsp[nip] o+ l+s+s2l+s+senl+s+s2p)
150     ninfopolyo.nwritep(l+s+s2+o.njoinp([n+nbstrp(npolyallp[nip]o.ncp[njp]) o+
    ↪  l+s+s2x o+ n+nbstrp(npolyallp[nip]o.norder o njp) kfor nj o+owin
    ↪  n+nbrangep(l+m+mi0p, npolyallp[nip]o.norder o+ l+m+mi1p)]) o+
    ↪  l+s+s2l+s+senl+s+s2p)
151     kfor nj o+owin n+nbrangep(l+m+mi0p, n+nblenp(nDataExtrap)):
152         ninfopolyo.nwritep(nfl+s+s2DataExtra[j]:
    ↪  str(polyall[i](DataExtra[j]))sl+s+senl+s+s2p)
153     ninfopolyo.nwritep(l+s+s2l+s+senl+s+s2p)
154 ninfopolyo.nclosep()
155
156 ngraficarp(nxrealallp[l+m+mi0p:l+m+mi7p], nyrealallp[l+m+mi0p:l+m+mi7p],
    ↪  nAlgorithmsp[l+m+mi0p:l+m+mi7p], l+s+s2Tamaño del problema (n)p, l+s+s2Tiempo
    ↪  (s)p,
157     l+s+s2Comparativa global de tiempos realesp,
    ↪  l+s+s2../graphics/globalComparativeTimes1.pngp, l+s+s1op)
158
159 ngraficarp(nxrealallp[l+m+mi0p:l+m+mi5p], nyrealallp[l+m+mi0p:l+m+mi5p],
    ↪  nAlgorithmsp[l+m+mi0p:l+m+mi5p], l+s+s2Tamaño del problema (n)p, l+s+s2Tiempo
    ↪  (s)p,
160     l+s+s2Comparativa global de tiempos realesp,
    ↪  l+s+s2../graphics/globalComparativeTimes2.pngp, l+s+s1op)
161
162 ngraficarp(nxrealallp[l+m+mi5p:l+m+mi7p], nyrealallp[l+m+mi5p:l+m+mi7p],
    ↪  nAlgorithmsp[l+m+mi5p:l+m+mi7p], l+s+s2Tamaño del problema (n)p, l+s+s2Tiempo
    ↪  (s)p,
163     l+s+s2Comparativa global de tiempos realesp,
    ↪  l+s+s2../graphics/globalComparativeTimes3.pngp, l+s+s1op)
164
165
166 ngraficarp(nxpolyallp[l+m+mi0p:l+m+mi7p], nypolyallp[l+m+mi0p:l+m+mi7p],
    ↪  nAlgorithmsp[l+m+mi0p:l+m+mi7p], l+s+s2Tamaño del problema (n)p, l+s+s2Tiempo
    ↪  (s)p,
167     l+s+s2Comparativa global de aproximaciones por polinomiop,
    ↪  l+s+s2../graphics/globalComparativePolynomial1.pngp, l+s+s1p)
168
169 ngraficarp(nxpolyallp[l+m+mi0p:l+m+mi5p], nypolyallp[l+m+mi0p:l+m+mi5p],
    ↪  nAlgorithmsp[l+m+mi0p:l+m+mi5p], l+s+s2Tamaño del problema (n)p, l+s+s2Tiempo
    ↪  (s)p,
170     l+s+s2Comparativa global de aproximaciones por polinomiop,
    ↪  l+s+s2../graphics/globalComparativePolynomial2.pngp, l+s+s1p)
171
172 ngraficarp(nxpolyallp[l+m+mi5p:l+m+mi7p], nypolyallp[l+m+mi5p:l+m+mi7p],
    ↪  nAlgorithmsp[l+m+mi5p:l+m+mi7p], l+s+s2Tamaño del problema (n)p, l+s+s2Tiempo
    ↪  (s)p,
```

```
173     l+s+s2Comparativa global de aproximaciones por polinomiop,
    ↪  l+s+s2../graphics/globalComparativePolynomial3.pngp, l+s+s1p)
```

.3. Compliación y ejecución

- **El script completo:**

  `python3.6 Make.py` (requiere las bibliotecas `matplotlib` y `numpy`).

- **Únicamente el programa principal:**

  ```
  gcc -std=c11 Time.c TestSortAlgorithms.c -o TestSortAlgorithms
  ./TestSortAlgorithms n NumAlgorithm OutputPlace <Input10Million.txt
  ```

Bibliografía

[1] S. Wikipedia and L. Books, *Sorting Algorithms: Sorting Algorithm, Merge Sort, Radix Sort, Insertion Sort, Heapsort, Selection Sort, Shell Sort, Bucket Sort*. General Books LLC, 2010. [Online]. Available: https://books.google.com.mx/books?id=VfU4bwAACAAJ