Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-2692 - Laboratorio de Algoritmos y Estructuras II
Trimestre Enero-Marzo 2016

# Estudio experimental de algoritmos de ordenamiento

## 1. Introducción

El objetivo del proyecto es el de hacer un estudio experimental de algoritmos de ordenamientos sobre diferentes conjuntos de datos. Algunos de los algoritmos que debe implementar son parte del *estado del arte* y son ampliamente usados. Una vez realizadas las pruebas solicitadas se quiere que haga un análisis de los resultados.

## 2. Variantes del algoritmos Quicksort

En esta sección se describen algunas variantes del algoritmo Quicksort, las cuales se van a estudiar en este proyecto.

### 2.1. *Median-of-3 quicksort*

Esta es una versión de Quicksort que toma como pivote la mediana de tres elementos. Los tres elementos a considerar en el arreglo son el primero, el último y el del medio. Este enfoque fue propuesto por [5] y tiene como característica la reducción del número esperado comparaciones con respecto al Quicksort original. En especifico se quiere que implemente el *median-of-3 quicksort* usado por la implementación de la función `sort` de la *Standard Template Library* (STL) de HP [6]. La Figura 1 muestra el pseudocódigo de este Quicksort. El procedimiento PARTITION es el que realiza la división de arreglo, y para ello debe hacer uso del método de partición propuesto por Hoare para el Quicksort original [2] y que se describe abajo. Como puede observar en la Figura 1, cuando se tiene que ordenar un número de elementos menor o igual a SIZE_THRESHOLD, entonces se usa el algoritmo INSERTION_SORT. El valor de SIZE_THRESHOLD a usar es 10.

```
PARTITION(A, p, r, x)
i = p − 1
j = r
while TRUE
    repeat
        j = j − 1
    until A[j] ≤ x
    repeat
        i = i + 1
    until A[i] ≥ x
    if i < j
        exchange A[i] with A[j]
    else return j
```

```
Algorithm QUICKSORT(A, f, b)
                    Inputs: A, a random access data structure containing the sequence
                        of data to be sorted, in positions A[f], ..., A[b − 1];
                      f, the first position of the sequence
                      b, the first position beyond the end of the sequence
                    Output: A is permuted so that A[f] ≤ A[f+1] ≤ ... ≤ A[b − 1]
    QUICKSORT_LOOP(A, f, b)
    INSERTION_SORT(A, f, b)

Algorithm QUICKSORT_LOOP(A, f, b)
                    Inputs: A, f, b as in QUICKSORT
                    Output: A is permuted so that A[i] ≤ A[j]
                      for all i, j: f ≤ i < j < b and size_threshold < j − i
    while b − f > size_threshold
        do   p := PARTITION(A, f, b, MEDIAN_OF_3(A[f], A[f+(b−f)/2], A[b−1]))
             if (p − f ≥ b − p)
                 then QUICKSORT_LOOP(A, p, b)
                      b := p
                 else QUICKSORT_LOOP(A, f, p)
                      f := p
```

Figura 1: *Median-of-3 quicksort* de la HP C++ STL, figura tomada de [3]

## 2.2. Introsort

Este algoritmo fue presentado por David Musser [3], y es una versión de Quicksort que limita la profundidad de la recursión. Cuando la profundidad de la recursión excede el límite establecido, entonces el algoritmo procede a ordenar los elementos restantes usando Heapsort. Esto hace que este algoritmo tenga un tiempo de ejecución en el peor caso de $O(N \log(N))$. Este algoritmo es ampliamente usado en la práctica, siendo el algoritmo de ordenamiento de la GNU C Library (glibc) [1], de la SGI C++ Standard Template Library [2] y de la Microsoft .NET Framework Class Library [3]. La Figura 2 muestra el pseudocódigo de INTROSORT. Se usará el mismo procedimiento PARTITION y el mismo valor de SIZE_THRESHOLD que se indicaron para la *Median-of-3 quicksort*. El valor del límite de profundidad $2 *$ FLOOR_LG(B-F) corresponde a $2\lfloor \log_2 K \rfloor$, donde $K$ es el número de elementos por ordenar del arreglo, es decir, el valor que se obtenga de $b - f$.

## 2.3. *Quicksort with 3-way partitioning*

Esta versión de quicksort usa el método de particionamiento propuesto por Bentley y McIlroy [1], el cual muy eficiente en la mayoría de los casos, incluyendo arreglos que están casi ordenados. Para ver en que consiste este particionamiento revise [4]. La Figura 3 muestra la variante de Quicksort presentada por Sedgewick y Bentley [4] que utiliza este particionamiento. Usted debe implementar el Quicksort de la Figura 3, haciendo la modificación que consiste, que en el caso de que tenga que para ordenar 10 o menos elementos del arreglo, entonces se debe usar el algoritmo de INSERTION_SORT.

[1] https://www.gnu.org/software/libc/
[2] http://www.sgi.com/tech/stl/
[3] https://msdn.microsoft.com/en-us/library/6tf1f0bc(v=vs.110).aspx

```
Algorithm Introsort(A, f, b)
                    Inputs: A, a random access data structure containing the sequence
                        of data to be sorted, in positions A[f], ..., A[b − 1];
                      f, the first position of the sequence
                      b, the first position beyond the end of the sequence
                    Output: A is permuted so that A[f] ≤ A[f+1] ≤ . . . ≤ A[b − 1]
    Introsort_loop(A, f, b, 2 * Floor_lg(b − f))
    Insertion_sort(A, f, b)

Algorithm Introsort_loop(A, f, b, depth_limit)
                    Inputs: A, f, b as in Introsort;
                      depth_limit, a nonnegative integer
                    Output: A is permuted so that A[i] ≤ A[j]
                      for all i, j: f ≤ i < j < b and size_threshold < j − i
    while b − f > size_threshold
        do if depth_limit = 0
                then Heapsort(A, f, b)
                    return
            depth_limit := depth_limit − 1
            p := Partition(A, f, b, Median_of_3(A[f], A[f+(b−f)/2], A[b−1]))
            Introsort_loop(A, p, b, depth_limit)
            b := p
```

Figura 2: Pseudocódigo de Introsort, figura tomada de [3]

```
void quicksort(Item a[], int l, int r)
{ int i = l-1, j = r, p = l-1, q = r; Item v = a[r];
  if (r <= l) return;
  for (;;)
    {
      while (a[++i] < v) ;
      while (v < a[--j]) if (j == l) break;
      if (i >= j) break;
      exch(a[i], a[j]);
      if (a[i] == v) { p++; exch(a[p], a[i]); }
      if (v == a[j]) { q--; exch(a[j], a[q]); }
    }
  exch(a[i], a[r]); j = i-1; i = i+1;
  for (k = l; k < p; k++, j--) exch(a[k], a[j]);
  for (k = r-1; k > q; k--, i++) exch(a[i], a[k]);
  quicksort(a, l, j);
  quicksort(a, i, r);
}
```

Figura 3: Pseudocódigo de *Quicksort with 3-way partitioning*, figura tomada de [4]

## 2.4. *Dual pivot Quicksort*

En 2009 Vladimir Yaroslavskiy propuso a la lista de correo de los desarrolladores de las librerías JAVA [4], una variante de Quicksort la cual usa dos pivotes. Este algoritmo fue

[4]http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628

escogida para ser el algoritmo de ordenamiento de arreglos por defecto de la librería de JAVA 7 de Oracle, después de un exhaustivo estudio experimental. También este algoritmo forma parte de la librerías de programación de la plataforma Android de Google [5]. Wild et al. [7] estudiaron esta versión de Quicksort y encontraron que su buen rendimiento se debe en buena parte a su método de particionamiento que hace que el algoritmo tenga en promedio menos comparaciones, que otras versiones de Quicksort. La Figura 4 muestra el pseudocódigo que debe implementar.

```
QuicksortYaroslavskiy(A, left, right)
    // Sort A[left, ..., right] (including end points).
    if right − left < M            // i.e. the subarray has n ≤ M elements
        InsertionSort(A, left, right)
    else
        if A[left] > A[right]
            p := A[right];   q := A[left]
        else
            p := A[left];    q := A[right]
        end if
        ℓ := left + 1;   g := right − 1;   k := ℓ
        while k ≤ g
            if A[k] < p
                Swap A[k] and A[ℓ]
                ℓ := ℓ + 1
            else
                if A[k] ≥ q
                    while A[g] > q and k < g do g := g − 1 end while
                    if A[g] ≥ p
                        Swap A[k] and A[g]
                    else
                        Swap A[k] and A[g];  Swap A[k] and A[ℓ]
                        ℓ := ℓ + 1
                    end if
                    g := g − 1
                end if
            end if
            k := k + 1
        end while
        ℓ := ℓ − 1;      g := g + 1
        A[left] := A[ℓ];      A[ℓ] := p       // Swap pivots to final position
        A[right] := A[g];     A[g] := q
        QuicksortYaroslavskiy(A, left, ℓ − 1)
        QuicksortYaroslavskiy(A, ℓ + 1, g − 1)
        QuicksortYaroslavskiy(A, g + 1, right)
    end if
```

Figura 4: Pseudocódigo del Dual pivot Quicksort, figura tomada de [7]

[5] https://goo.gl/5GIQd1

## 3. Conjunto de datos a evaluar

Se desea probar los algoritmos de ordenamiento con diferentes conjuntos de datos. Suponemos que la meta es ordenar los arreglos de forma ascendente. Cada conjunto de datos corresponde a una prueba en la cual el arreglo tiene diferente contenido. A continuación se presenta los conjuntos de datos a utilizar:

1. **Enteros aleatorios:** Números enteros comprendidos en el intervalo $[0, 500]$ generados aleatoriamente.

2. **Orden inverso:** Si el tamaño del arreglo es $N$, entonces el arreglo contendrá la secuencia $N, N-1, \ldots, 2, 1$.

3. **Cero-uno:** Ceros y unos generados aleatoriamente.

4. **Ordenado:** Si el tamaño del arreglo es $N$, entonces el arreglo contendrá la secuencia $1, 2, \ldots, N-1, N$.

5. **Reales aleatorios:** Números reales comprendidos en el intervalo $[0, 1)$ generados aleatoriamente.

6. **Mitad:** Dado un arreglo de tamaño $N$, el arreglo contiene como elementos la secuencia de la forma $1, 2, \ldots, N/2, N/2, \ldots, 2, 1$.

7. **Casi ordenado:** Dado un conjunto ordenado de $N$ elementos de tipo entero, se escogen al azar $n/4$ pares de elementos que se encuentran separados 4 lugares, entonces se intercambian los pares.

## 4. Actividades a realizar

El código base para la realización de proyecto va a ser el código que fue usado en el Laboratorio de la semana 5. El código debe contener las actividades requeridas en dicho laboratorio. El código lo debe conformar los siguientes archivos:

- `arrayT.py`: Módulo con la definición de arreglos a utilizar.

- `ordenamiento.py`: Módulo que debe contener la implementación de los algoritmos de ordenamientos.

- `cliente_ordenamiento.py`: Cliente que ejecuta los algoritmos de ordenamiento.

A continuación se presentan las actividades que debe llevar a cabo.

### 4.1. Creación del conjunto de datos

En el Laboratorio de la Semana 5 se les solicitó modificar el archivo `cliente_ordenamiento.py`, para incorporar en los argumentos de la línea de comandos la opción `-p` y `--prueba`, para escoger con cual conjunto de datos a trabajar. Tenga en cuenta que los tres conjuntos de datos que se deben haber implementado hasta el Laboratorio de la semana 5, corresponden a los conjuntos de datos descritos en la Sección 3 como **Enteros aleatorios**, **Orden inverso** y **Cero-uno**. En consecuencia, debería adicionalmente implementar el resto del conjunto de datos indicados en la Sección 3, tal que el número que identifique a cada conjunto de datos sea el mismo con el que se numera su definición. Por ejemplo, si se quiere ejecutar el conjunto de datos **Mitad**, sobre arreglos de tamaño 500 y 1000, ejecutándose cada algoritmo 2 veces sobre cada uno de los arreglos, entonces la siguiente línea de comando es válida:

```
>./cliente_ordenamiento.py -p 6 -t 2 500 1000
```

Otro ejemplo, la línea de comando de abajo es similar a la anterior, pero se corren los algoritmos de ordenamiento sobre el conjuntos de datos llamada **Reales aleatorios**.

```
>./cliente_ordenamiento.py -p 5 -t 2 500 1000
```

## 4.2. Carga de los algoritmos de ordenamiento

El programa `cliente_ordenamiento.py` carga todos los procedimientos del módulo `ordenamiento.py`. Usted debe cambiar esto, para que cargue únicamente los procedimientos que corresponde a los 7 algoritmos de ordenamiento a implementar.

Este modificación hace posible que el modulo `ordenamiento.py`, contenga todos los procedimientos asociados a cada algoritmo de ordenamiento.

## 4.3. Implementación de los algoritmos de ordenamiento

Se quiere que implemente los siguientes 7 algoritmos de ordenamiento:

1. **Mergesort:** Debe implementar la versión de Mergesort requerida en el laboratorio de la semana 3.
2. **Heapsort:** Debe implementar la versión Heapsort requerida en el laboratorio de la semana 4.
3. **Quicksort aleatorio:** Debe implementar la versión Quicksort aleatorio requerida en el laboratorio de la semana 5.
4. ***Median-of-3 quicksort***
5. **Introsort**
6. ***Quicksort with 3-way partitioning***
7. ***Dual pivot Quicksort***

Todos los algoritmos de ordenamiento, y sus procedimientos auxiliares, deben estar implementados en el módulo `ordenamiento.py`. En la implementación de los algoritmos de ordenamientos, *no debe hacer uso de las listas de Python.* Los arreglos que deben utilizar los algoritmos de ordenamientos son los representados en el módulo `arrayT.py`. Debido a esto, el código que usted debe entregar debe estar compuesto únicamente de 3 archivos de `Python 3`: `arrayT.py`, `ordenamiento.py` y `cliente_ordenamiento.py`.

### 4.4. Estudio experimental a realizar

Los 7 algoritmos de ordenamiento se deben ejecutar en cada uno de los 7 conjuntos de datos. Para cada conjunto de datos se debe hacer la corrida en arreglos de tamaño 4096, 8192, 16384, 32768 y 65536. La ejecución de un algoritmo en un arreglo de tamaño determinado la llamamos prueba. Cada prueba debe ejecutarse 3 veces y se reportará como tiempo de ejecución del algoritmo, el tiempo promedio de esas tres pruebas. Es decir, una prueba es el resultado que se obtiene al hacer uso de la opción `-t 3` del programa `cliente_ordenamiento.py`.

La ejecución de algunas de las pruebas puede necesitar que aumente el límite del número de recursiones permitdas por el interpretador de Python. Aumente ese límite de manera tal, que todas las pruebas requeridas puedan ser llevadas a cabo.

### 4.5. Análisis de los resultados

Debe realizar un informe con los resultados obtenidos y el análisis de los mismo. En primer lugar, debe describir el tipo de equipo en donde se realizaron los experimentos, indicando el modelo de CPU, la cantidad de memoria RAM y el sistema de operación usado. Luego, por cada uno de los 7 conjunto de datos debe realizar una tabla con los resultados obtenidos. Cada tabla debe tener el formato mostrado para la Tabla 1

| **N** | **Mergesort** | **Heapsort** | **QS Random** | **Med-of-3 QS** | **Introsort** | **Dual pivot QS** | **QS 3-way** |
|---|---|---|---|---|---|---|---|
| 4096 | | | | | | | |
| 8192 | | | | | | | |
| 16384 | | | | | | | |
| 32768 | | | | | | | |
| 65536 | | | | | | | |

Tabla 1: Resultados del tiempo de ejecución de la conjunto de datos X, en segundos

Por cada tabla de resultados, en el informe debe hacer un análisis de los resultados obtenidos y una gráfica de los resultados de la tabla, en la que se muestren los puntos de todos los algoritmos de ordenamiento y la tendencia de esos puntos. En caso en que uno o más algoritmos presenten un orden de crecimiento cuadrático, entonces debe realizar una segunda corrida solo con los algoritmos que presenten un orden de de crecimiento lineal y cuasi-lineal y debe mostrar en el informe la gráfica resultante. Después de haber analizado cada una de las pruebas realizadas, debe hacer un análisis global de todos los resultados obtenidos. Se quiere que haga mención, entre otros aspectos, si los resultados se ajustan a lo esperado de la teoría y también ¿cuáles algoritmos son los que presentan mejor y peor comportamiento? El informe debe estar en formato PDF.

## 5. Condiciones de entrega

Su programa debe poderse ejecutarse en `Python 3` en el sistema de operación `Linux`. Si un programa no se ejecuta, el equipo tiene cero como nota del proyecto.

El trabajo es por equipos de laboratorio. Debe entregar los códigos fuentes de sus programas y el informe, en un archivo comprimido llamado `Proyecto1-X-Y.tar.gz` donde X y Y son los números de carné de los integrantes del grupo. La entrega se realizará por medio del aula virtual **antes** de la 1:00 pm del viernes 24 de Febrero de 2017. Cada grupo debe entregar firmada, en el laboratorio de la semana 8, la "Declaración de Autenticidad para Entregas", cuya plantilla se encuentra en la página del curso. Ambos integrantes del equipo deben trabajar en el proyecto. El no cumplimiento de algunos de los requerimientos podrá resultar en el rechazo de su entrega.

---

Guillermo Palma / gvpalma@usb.ve / Febrero de 2017

## Referencias

[1] Bentley, J. L., and McIlroy, M. D. Engineering a sort function. *Software: Practice and Experience 23*, 11 (1993), 1249–1265.

[2] Hoare, C. A. Quicksort. *The Computer Journal 5*, 1 (1962), 10–16.

[3] Musser, D. R. Introspective sorting and selection algorithms. *Softw., Pract. Exper. 27*, 8 (1997), 983–993.

[4] Sedgewick, R., and Bentley, J. Quicksort is optimal. `http://www.sorting-algorithms.com/static/QuicksortIsOptimal.pdf`, January 2002.

[5] Singleton, R. C. Algorithm 347: an efficient algorithm for sorting with minimal storage [m1]. *Communications of the ACM 12*, 3 (1969), 185–186.

[6] Stepanov, A., and Lee, M. The standard template library. `http://www.hpl.hp.com/techreports/95/HPL-95-11.html`, 1995.

[7] Wild, S., Nebel, M. E., and Neininger, R. Average case and distributional analysis of dual-pivot quicksort. *ACM Transactions on Algorithms (TALG) 11*, 3 (2015), 22.