Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-2692 - Laboratorio de Algoritmos y Estructuras de Datos II
Trimestre enero-marzo 2020

Estudio experimental de algoritmos de ordenamiento

(Versión 0.1)

1. Introducción

El objetivo del proyecto es el de hacer un estudio experimental de algoritmos de ordenamientos, simples y de alto rendimiento, sobre diferentes tipos de secuencias. La idea es comparar el rendimiento de los algoritmos de ordenamiento vistos en los libros de texto, junto con otras versiones que son el *estado del arte* y que son ampliamente usados en la práctica. Una vez hechas las pruebas experimentales de los algoritmos de ordenamiento, se quiere que haga un análisis de los resultados.

2. Algoritmos de ordenamiento

En esta sección se describen los algoritmos de ordenamiento que van a ser objeto de estudio. A continuación se hace una breve descripción de cada uno de ellos.

Mergesort: versión de Mergesort presentada en la página 229 de [2]. Cuando el número de elementos a ordenar es menor o igual a 32, entonces se ordena la secuencia con el algoritmo INSERTION_SORT.

Quicksort iterativo: versión de Quicksort presentada en la página 179 de [6].

Quicksort simple: versión de Quicksort presentada en la página 171 de [3], con la modificación de que si la secuencia o subsecuencia de entrada tiene un número de elementos a ordenar que es menor o igual a 32, entonces se ordena la mismo con el algoritmo INSERTION_SORT.

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 [10] 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 [11]. 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 procedimiento de partición propuesto por Hoare para el algoritmo Quicksort original [5], y que se muestra en la Figura 2. 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 32.

Introsort: Este algoritmo fue presentado por David Musser [7], 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

```
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. Fuente [7].

```
PARTITION(A, p, r, x)
 1  i = p − 1
 2  j = r
 3  while TRUE
 4       repeat
 5            j = j − 1
 6       until A[j] ≤ x
 7       repeat
 8            i = i + 1
 9       until A[i] ≥ x
10       if i < j
11            exchange A[i] with A[j]
12       else return j
```

Figura 2: Procedimiento de partición propuesto por Hoare para el algoritmo Quicksort.

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 3 muestra el pseudocódigo del algoritmo INTROSORT. Se usará el mismo procedimiento PARTITION de la Figura 2 y un valor de SIZE_THRESHOLD de

[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

32. El valor del límite de profundidad $2 * \text{FLOOR\_LG}(\text{B-F})$ 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 $b - f$.

```
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 3: Pseudocódigo del algoritmo INTROSORT. Fuente [7].

***Quicksort with 3-way partitioning*:** Esta es una versión de Quicksort presentada por Sedgewick y Bentley [9] que utiliza el procedimiento de partición de un arreglo propuesto por Bentley y McIlroy [1]. Este procedimiento es eficiente en la mayoría de los casos, incluyendo arreglos que están casi ordenados. Una explicación de este procedimiento de partición se encuentra en [9]. La figura 4 muestra esta versión de Quicksort. Usted debe implementar el Quicksort de la figura [9], haciendo la modificación que consiste, que en el caso de que tenga que para ordenar una secuencia con un número de elementos igual o menor a 32, entonces se debe usar el algoritmo de INSERTION_SORT.

***Dual pivot Quicksort*:** En 2009 Vladimir Yaroslavskiy propuso a la lista de correo de los desarrolladores de las librerías JAVA [4], una versión de Quicksort la cual usa dos pivotes. Este algoritmo fue escogido para ser el algoritmo de ordenamiento de arreglos por defecto de la librería de JAVA 7 de Oracle, después de un exhaustivo estudio experimental. Wild et al. [12] estudiaron esta versión de Quicksort y encontraron que su buen rendimiento se debe en buena parte a su procedimiento de partición hace que el algoritmo tenga en promedio menos comparaciones, que otras versiones de Quicksort. La Figura 5 muestra el pseudocódigo que debe implementar. Se puede observar en las líneas 1 y 2 de la Figura 5, que si la secuencia o subsecuencia a

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

```
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 4: Pseudocódigo del algoritmo *Quicksort with 3-way partitioning*. Fuente [9].

ordenar tiene un número de elementos menor o igual a M, entonces usa el algoritmo de INSERTION_SORT. El valor de M que debe usar es 32.

***Timsort*:** Es un algoritmo de ordenamiento estable propuesto por Tim Peters [8] en el 2002. TIMSORT está basado en MERGESORT e INSERTION_SORT. Timsort tiene un tiempo de $O(n\log(n))$ en el peor caso y en el caso promedio, mientras que es $O(n)$ en el mejor caso. Asimismo usa un espacio de $O(n/2)$ en el peor caso y constante en el mejor caso. En la práctica TIMSORT ha resultado un algoritmo muy rápido sobre todo tipo de datos. Esto ha llevado a que TIMSORT sea el algoritmo de ordenamiento estándar utilizado por Python [5]. También TIMSORT es usado en JAVA OpenJDK [6], para ordenar arreglos de elementos que no son de tipos primitivos, GNU Octave [7], y Google Chrome [8], entre otros. Para este proyecto no va a ser necesario implementar TIMSORT. Se va a utilizar la implementación en Python del algoritmo TIMSORT presentada por Karthik Desingu [4].

Debe crear una librería llamada `orderlib.py` que contenga todos los algoritmos presentados en esta sección.

3. Programa de pruebas

Se quiere que implemente un programa, llamado `prueba_orderlib.py` para probar todos los algoritmos de ordenamiento de la librería `orderlib.py`. El programa debe ejecutar varias pruebas sobre todos los algoritmos de ordenamiento de la librería. Las pruebas tienen diferentes tipos de secuencias que van a recibir los algoritmos de ordenamiento. A continuación se presenta las secuencias de datos de entrada a utilizar:

[5] https://bit.ly/2vT1pCb
[6] https://bugs.openjdk.java.net/browse/JDK-6804124
[7] https://bit.ly/3bdoSOF
[8] https://bit.ly/2SnG5vY

```
QUICKSORTYAROSLAVSKIY(A,left,right)
    // Sort A[left,...,right] (including end points).
 1  if right − left < M            // i.e. the subarray has n ≤ M elements
 2      INSERTIONSORT(A,left,right)
 3  else
 4      if A[left] > A[right]
 5          p := A[right];   q := A[left]
 6      else
 7          p := A[left];    q := A[right]
 8      end if
 9      ℓ := left + 1;   g := right − 1;   k := ℓ
10      while k ≤ g
11          if A[k] < p
12              Swap A[k] and A[ℓ]
13              ℓ := ℓ + 1
14          else
15              if A[k] ≥ q
16                  while A[g] > q and k < g do g := g − 1 end while
17                  if A[g] ≥ p
18                      Swap A[k] and A[g]
19                  else
20                      Swap A[k] and A[g];  Swap A[k] and A[ℓ]
21                      ℓ := ℓ + 1
22                  end if
23                  g := g − 1
24              end if
25          end if
26          k := k + 1
27      end while
28      ℓ := ℓ − 1;        g := g + 1
29      A[left] := A[ℓ];       A[ℓ] := p      // Swap pivots to final position
30      A[right] := A[g];      A[g] := q
31      QUICKSORTYAROSLAVSKIY(A, left , ℓ − 1)
32      QUICKSORTYAROSLAVSKIY(A, ℓ + 1,g − 1)
33      QUICKSORTYAROSLAVSKIY(A,g + 1,right)
34  end if
```

Figura 5: Pseudocódigo del algoritmo Dual pivot Quicksort. Fuente [12]

1. **Punto flotante**: Números reales comprendidos en el intervalo $[0, 1)$ generados aleatoriamente.

2. **Ordenado**: Número enteros ordenados en orden ascendente.

3. **Orden inverso**: Número enteros que se encuentran ordenados descendentemente.

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

5. **Mitad**: Dado una secuencia de tamaño N, la secuencia de entrada contiene elementos con la forma $1, 2, \ldots, N/2, N/2, \ldots, 2, 1$.

6. **Casi ordenado 1**: Dado un conjunto ordenado de elementos de tipo entero, se escogen al azar 16 pares de elementos que se encuentran separados 8 lugares, entonces se intercambian los pares.

7. **Casi ordenado 2**: 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.

Para ordenar las secuencias de entrada de los algoritmos de ordenamiento, se puede usar los métodos *sort* y *sorted* de Python.

El usuario puede seleccionar la prueba a aplicar y el número de veces, o intentos, que se va aplicar esa prueba sobre los algoritmos de ordenamiento. También el usuario **puede indicar una lista con el tamaño de la secuencias de entrada** a ordenar. El resultado del programa **es el tiempo promedio de ejecución y la desviación estándar, de cada uno de los algoritmos** sobre las secuencias generadas de un tipo específico. Es importante notar que una vez escogida un tipo de secuencia, **en cada nuevo intento se debe generar una nueva secuencia, y esta misma nueva secuencia debe ser ordenada por todos los algoritmos de ordenamientos seleccionados**. De esta forma se puede comparar el tiempo de cada uno de los algoritmos de ordenamiento, sobre la misma secuencia de entrada. Se quiere que el programa de pruebas **pueda graficar los tiempos de ejecución de los algoritmos de ordenamiento, dada la lista de tamaños de secuencias de entrada**. Para graficar los resultados de los algoritmos de ordenamiento debe hacer uso de la librería `graficar_puntos.py`. Esta librería permite tener un en solo gráfico, las curvas del tiempo de ejecución de varios algoritmos de ordenamiento. De esta manera es posible ver gráficamente el desempeño de los algoritmos de ordenamiento, y determinar cual es más rápido para un tipo de entrada específica

La ejecución de `prueba_orderlib.py` se hace por medio de la siguiente línea de comando:

```
>./prueba_orderlib.py [-i #num] [-t #num] [-g] tamaño_1 .. tamaño_N
```

La semántica de los parámetros de entrada es la siguiente:

- **-t** Prueba a realizar. Los valores posibles están en el intervalo $[1, 7]$. Cada valor corresponde a una de las pruebas descrita anteriormente.
- **-i**: Número de pruebas o intentos a realizar de la prueba seleccionada.
- **tamaño_1 .. tamaño_N** lista con los números de elementos, o tamaño, de la secuencias de entradas que van a ser ordenadas.
- **-g**: indica que van a ser graficados los resultados, de todas los diferentes tamaños de secuencias ejecutados. Esta opción solo es válida si hay dos o más tamaños de secuencia. En caso contrario, el programa aborta con mensaje de error.

Cuando se aplique la opción **-g**, primero el programa debe mostrar todos los resultados por la salida estándar y luego, se debe mostrar la gráfica resultante. Es obligatorio chequear que debe haber al menos un valor de tamaño de la secuencia. Todos los demás argumentos de la línea de comandos son opcionales. Para manejar los argumentos de entrada del programa, debe hacer uso del módulo argparse o del módulo getopt. Los valores por de defecto son:

- **-t**: 1, es decir, el tipo de secuencia de entrada, es la generada por el tipo 1.
- **-i**: 3, es decir, se ejecutará tres veces sobre los algoritmos de ordenamiento la prueba seleccionada.

- **-g**: false, es decir, el programa no mostrará las gráficas del tiempo de ejecución.

```
>./prueba_ord.py  -t 2 -i 6  1000 2000 3000
```

El programa ejecutará la prueba 2. En esta prueba se ejecutará con tres secuencias que contienen 1000, 2000 y 3000 elementos. Para cada tamaño de secuencia se generarán seis secuencias diferentes. Se ejecutarán todos los algoritmos de ordenamiento del módulo `orderlib`. sobre las seis secuencias generadas, para cada uno de los tres tamaños de secuencia.

La salida del programa es tiempo **promedio y desviación estándar**, de cada uno de los algoritmos de ordenamiento ejecutados, **los cuales deben estar en una misma línea para su mejor lectura**.

4. Estudio experimental

Debe ejecutar todos los algoritmos de ordenamiento del módulo `orderlib` sobre todos los tipos de secuencia. Se ejecutarán secuencias de tamaño 4096, 8192, 16384, 32768 y 65536. Cada secuencia debe ser ejecutada tres veces. Al final de la ejecución se debe mostrar la gráfica de los tiempos de ejecución de los algoritmos de ordenamiento, para cada tamaño de secuencia. Es decir, el programa debe ser ejecutado con la siguiente línea de comando:

```
>./prueba_ord.py  -t X -i 3  -g 4096  8192 16384 32768  65536
```

Donde X se sustituye por el tipo de prueba a realizar. El objetivo es poder observar claramente en una gráfica el crecimiento del tiempo de ejecución de los algoritmos, tal que, sea posible distinguir la tendencia de crecimiento y determinar cual de los algoritmos de ordenamiento, presenta el mejor desempeño.

Debe realizar un informe que sólo debe contener los resultados obtenidos y el análisis de los mismo. En específico, el informe debe contener lo siguiente:

1. Descripción de la plataforma utilizada para correr los experimentos, es decir, sistema de operación, modelo de CPU y memoria RAM del computador.

2. Siete tablas, una tabla por cada tipo de secuencia. Cada tabla debe contener el tiempo promedio todos los algoritmos de ordenamiento para los tamaños de secuencia usados. cada tabla debe tener el formato mostrado para la Tabla 1

3. La gráfica de tiempo de ejecución correspondiente para cada tabla de resultados. En caso en que uno o más algoritmos presenten un orden de crecimiento cuadrático, entonces debe realizar una segunda gráfica en donde sólo se muestren los algoritmos que presenten un orden de de crecimiento lineal y cuasi-lineal. Puede hacer uso del programa `graficar_puntos.py`, que se ha usado en los laboratorios anteriores.

4. Debe realizar un análisis de los resultados obtenidos, indicando entre otros aspectos, si los mismos se ajustan a lo esperado de la teoría y ¿cuáles algoritmos son los que presentan mejor y peor comportamiento?

5. El informe debe estar en formato PDF.

N	Mergesort	QS iter	QS simple	Med-of-3	Introsort	3-way	DP	Timsort
4096								
8192								
16384								
32768								
65536								

Tabla 1: Ejemplo de la tabla con los resultados del tiempo de ejecución

5. Condiciones de entrega

El trabajo es por equipos de máximo dos personas. El código a entregar debe estar debidamente documentado y para todos los procedimientos debe indicar su descripción, las pre y post condiciones, y la explicación de los parámetros de entrada. Las implementaciones de los algoritmos de ordenamiento, deben seguir los pseudocódigos aquí indicados. Se debe seguir los lineamientos de la guía de estilo de Python. El programa que entregue debe poderse ejecutarse en Linux. Si un programa no se ejecuta, el equipo tiene cero como nota del proyecto.

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 equipo. La entrega del archivo `Proyecto1-X-Y.tar.gz`, debe hacerse al profesor del laboratorio por email, antes de las 11:50 pm del día 19 de febrero de 2020.

Referencias

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

[2] Brassard, G., and Bratley, P. *Fundamentals of Algorithmics*. Prentice Hall, 1996.

[3] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. *Introduction to algorithms*, 3rd ed. MIT press, 2009.

[4] Desingu, K. Timsort algorithm implementation in python. `https://www.codespeedy.com/timsort-algorithm-implementation-in-python/`, February 2020.

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

[6] Kaldewaij, A. *Programming: the derivation of algorithms*. Prentice-Hall, Inc., 1990.

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

[8] Peters, T. Timsort. `http://svn.python.org/projects/python/trunk/Objects/listsort.txt`, February 2011.

[9] Sedgewick, R., and Bentley, J. Quicksort is optimal. `https://gpalma.github.io/courses/ci2692em20/QuicksortIsOptimal.pdf`, January 2002.

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

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

[12] 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.

Guillermo Palma / gvpalma@usb.ve / Febrero de 2020