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

Proyecto: Estudio experimental de algoritmos de ordenamiento (versión 1.1)

1. Introducción

El objetivo del proyecto es el de hacer un estudio experimental de algoritmos de ordenamientos sobre arreglos. Se quiere ver el rendimiento del algoritmo Heapsort y cuatro versiones del algoritmo de Quicksort, en varios escenarios de pruebas. 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. Algoritmos de ordenamiento a estudiar

A continuación se describen los algoritmos que debe implementar.

Heapsort: La versión que se presenta en el página 160 del Cormen et al. [2].

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 [6] 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 [7]. 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 [3] 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 15.

```
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
```

```
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 [4]

Introsort: Este algoritmo fue presentado por David Musser [4], y es una versión de Quicksort que limita la profundidad de la recursión del particionamiento. 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 $2 * \text{FLOOR\_LG(B-F)}$ 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 $b - f$.

***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 [5]. La figura 3 muestra la variante de Quicksort presentada por Sedgewick y Bentley [5] que utiliza este particionamiento. Usted debe implementar el Quicksort de la figura [5], haciendo la modificación que consiste, que en el caso de que tenga que para ordenar 15 o menos elementos del arreglo, entonces se debe usar el algoritmo de INSERTION_SORT.

***Dual pivot Quicksort*:** En 2009 Vladimir Yaroslavskiy propuso a la lista de correo 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

```
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 [4]

```
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 [5]

los desarrolladores de las librerías JAVA [4], una variante de Quicksort la cual usa dos pivotes. Este algoritmo fue 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

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

la plataforma Android de Google [5]. Wild et al. [8] 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).
 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 4: Pseudocódigo del Dual pivot Quicksort, figura tomada de [8]

3. Evaluación experimental

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

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

presenta los conjuntos de datos a utilizar:

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

Ordenado: Número enteros ordenados en orden ascendente.

Orden inverso: Número enteros que se encuentran ordenados descendentemente.

Cero-uno: Ceros y unos generados aleatoriamente.

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

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.

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.

Cada uno de los 7 conjuntos de datos se deben ejecutar en arreglos de tamaño 4096, 8192, 16384, 32768, 65536 y 131072. La ejecución de un algoritmo en un arreglo de tamaño determinado la llamamos prueba. Cada prueba debe ejecutarse 10 veces, de estas 10 se eliminan el mejor y el peor tiempo obtenido y se reportará como tiempo de ejecución del algoritmo, el promedio de las 8 pruebas restantes.

4. Detalles de la implementación

Debe implementar el proyecto en el lenguaje de programación Python version 2.7. Debe crear un programa llamado `prueba_proyecto1.py` que se ejecuta con la siguiente llamada:

```
>python prueba_proyecto1 <numero de elementos> <dataset> <numero de pruebas>
```

donde el parámetro < numero de elementos > es el número de elementos que va a tener el arreglo a ordenar, < dataset > es el número del conjunto de datos a ejecutar, dada la numeración descrita anteriormente, y < numero de pruebas > es la cantidad de pruebas a realizar. Tenga en cuenta que como se elimina el peor y el mejor tiempo de cada prueba, entonces el menor número de pruebas permitido es 3. Un ejemplo de la llamado de la línea de comando sería:

```
>python prueba_proyecto1 4096 7 10
```

en cuyo caso se ejecutarían todos los algoritmos de ordenamiento 10 veces en un arreglo de tamaño 4096, para el conjunto de datos *Casi ordenado 2*. La salida del programa `prueba_proyecto1` debe mostrar el tiempo promedio, en segundos, obtenido por cada algoritmo.

EL código de debe estar debidamente documentado y para procedimientos debe indicar su descripción, las pre y post condiciones, y explicación de los parámetros de entrada. Las implementaciones de los algoritmos de ordenamiento, deben seguir los pseudocódigos aquí indicados.

5. Condiciones de entrega

Debe realizar un informe que sólo debe contener 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 conjunto de datos debe realizar una tabla con los resultados obtenidos, cada tabla debe tener el formato mostrado para la tabla 1

N	Heapsort	Med-of-3 QS	Introsort	Dual pivot QS	QS with 3-way
4096					
8192					
16384					
32768					
65536					
131072					

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

Por cada tabla de resultados, en el informe debe haber 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 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. Por último, debe realizar un análisis de los resultados obtenidos, indicando entre otros aspectos, si los mismos se ajustan a lo esperado de la teoría, ¿qué algoritmos presentan un orden de ejecución cuadrático?, ¿cuáles algoritmos son los que presentan mejor y peor comportamiento? El informe debe estar en formato PDF.

El programa que entregue debe poderse ejecutarse en 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 12:50 pm del sábado 20 de Febrero de 2016. Cada grupo debe entregar firmada a su encargado de laboratorio, 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.

Referencias

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

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

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

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

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

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

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

[8] 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 en usb.ve / Febrero 2016