Práctica A2

Ordenamiento

En general, cuando tenemos que comparar el rendimiento de dos algoritmos, debemos considerar sus mejores y peores casos. Los mismos dependen de cada algoritmo, pero un buen conjunto para comenzar es:

- Arreglo creciente
- Arreglo decreciente
- Arreglo constante
- Arreglo aleatorio

Los ejercicios marcados con un asterisco son difíciles y pueden saltearse hasta terminar todo el resto.

Implementación

1. ¿Cómo cambia el problema de encontrar el máximo si trabajamos con arreglos ordenados? Ídem mínimo y mediana.

2. Implemente búsqueda binaria sobre un arreglo ordenado de enteros. La misma deberá tener el tipo:

```
int binsearch(int a[], int len, int v)
```

y devolver el índice i en donde se encuentra el valor v, o -1 si no aparece en el array.

3. Implemente la mejora discutida en clase a Insertion sort. Es decir, evite los `swap()`s intermedios innecesarios. Mida la diferencia de rendimiento: ¿cuándo es más pronunciada?

4. Implemente Mergesort para arreglos, haciendo un manejo adecuado de la memoria.

5. Implemente Quicksort con la partición de Lomuto como se vio en clase. Compare el rendimiento contra Mergesort, tanto en arreglos aleatorios como en sus peores casos.

6. Modifique la partición de Lomuto para llevar tres segmentos: menores estrictos al pivote, iguales al pivote, y mayores estrictos al pivote. Usándola, adapte Quicksort para funcionar eficientemente en arrays con muchos elementos repetidos.

7. Diseñe un algoritmo que dado un arreglo A de tamaño n calcule la mediana del mismo. Puede modificar al arreglo que recibe como argumento. (Pista: la mediana es el elemento medio del arreglo ordenado.) Su función debería ser mejor que $O(n \lg n)$ en el caso promedio.

8. Generalice el ejercicio anterior para encontrar el k-ésimo elemento del arreglo ordenado.

9. Generalice el ejercicio anterior para encontrar los k primeros elementos del arreglo ordenado.

10. Compare los tiempos de ejecución de Insertion sort y Quicksort a medida que crece la cantidad de elementos. Hágalo al menos para arreglos crecientes, decrecientes, y aleatorios.

11. Necesitamos ordenar un array de $N - 1$ enteros (sin datos extra) que tiene solamente números entre 1 y N, sin repetir. Diseñe una función que lo haga en tiempo lineal.

Adicionales

Propiedades de los algoritmos de ordenamiento

12. ¿Cuáles de los algoritmos presentados en clase son estables? Para ellos, verifique que su implementación es realmente estable.

13. El `qsort()` provisto por la librería estándar de C no es estable. Imagine que tiene que ordenar elementos de la siguiente estructura por el campo `grupo`, pero necesita hacerlo de manera estable:

```
struct persona {
    int grupo;
    char nombre[256];
}
```

Busque una manera de usar `qsort()` para lograr ordenar de manera estable. Puede modificar la estructura o crear nuevas. Dado que pudo hacer esto, ¿por qué nos preocupamos por la estabilidad?

14*. Escriba una variante adaptiva de Mergesort. Pista: partir desde una versión "bottom-up".

Maximizando el rendimiento

15. Modifique su versión de Quicksort para usar Insertion sort cuando el arreglo tiene tamaño menor a algún umbral K[1]. Optimice el valor de K midiendo la diferencia en performance.

16*. En cada paso de la búsqueda binaria, se hacen dos comparaciones del entero buscado con el elemento del array (una por igualdad, una por orden). En el peor caso, se hacen en total $2 \times \lceil \lg_2 n \rceil$ comparaciones. Escriba una versión que reduzca este número a $\lceil \lg_2 n \rceil$ en el peor caso, tal vez agregando una o dos comparaciones adicionales. Pista: adaptar la búsqueda para encontrar el último índice i tal que $a[i] \leq v$. Si tenemos ese índice, ¿cómo sabemos si v está en el array?

17.** En cada inserción de Insertion sort, se busca el lugar en donde insertar el elemento recorriendo el array linealmente. Implemente una versión que busca esta posición usando búsqueda binaria. Luego, puede moverse el subarray usando `memmove()` de la librería estándar de C. ¿Cuáles son ahora las complejidades en mejor y peor caso? ¿Qué podemos esperar del caso promedio? Mida la diferencia de performance.

18 (Partición de Hoare). En la versión original de Quicksort presentada por Hoare (en 1961) la función de partición estaba implementada de otra manera. La idea general es recorrer el arreglo desde ambos extremos, intercambiando elementos cuando están invertidos respecto al pivote (es decir, con $i < j$, tenemos $a[i] > p > a[j]$). El pseudocódigo de esta versión es:

function PARTITION(a, n, p)
  $i \leftarrow 0$
  $j \leftarrow n - 1$
  while TRUE **do**
    while $i < n \wedge a[i] < p$ **do**
      $i \leftarrow i + 1$
    end while
    while $j \geq 0 \wedge a[j] > p$ **do**
      $j \leftarrow j - 1$
    end while

[1] Otra opción (atribuída a Robert Sedgewick) es que Quicksort no haga nada en arreglos más chicos que K, dejando el arreglo "casi" ordenado. Luego se hace una pasada final con Insertion sort sobre todo el arreglo.

```
        if i ≥ j then
            return i
        end if
        intercambiar a[i] ↔ a[j]
        i ← i + 1; j ← j − 1
    end while
end function
```

Esta versión suele ser más eficiente que la de Lomuto, pero es notoriamente engañosa de implementar correctamente. Asegúrese de entenderla (ej: ¿siempre termina? ¿qué pasa con los elementos iguales al pivote?) e implemente Quicksort con la partición de Hoare.

19. Imagine que tenemos datos sobre el uso del sistema de bicicletas públicas de Rosario en un arreglo. Cada elemento contiene un viaje, y el arreglo está ordenado por la fecha y hora del *comienzo* del viaje. Necesitamos ahora ordenar el arreglo por la fecha y hora de *finalización*. Como los viajes duran siempre menos de 30 minutos, el arreglo ya está "casi" ordenado. Justifique que es esperable que Insertion sort sea más rápido que Quicksort en este caso.

Análisis de algoritmos

20. Una *inversión* en un arreglo A es un par (i, j) con $i < j$ y $A[i] > A[j]$. Ej: un array ordenado tiene 0 inversiones, y un array ordenado al revés tiene $\frac{n(n-1)}{2}$. Demuestre que Bubble sort siempre hace una cantidad de swaps igual a la cantidad de inversiones. (Alternativa: escriba una versión de Bubble sort que cuente los swaps que realiza y una función que cuente las inversiones en un array, y ejecútelas sobre arreglos grandes.)

21. En clase demostramos que si Quicksort divide el arreglo en proporción $\frac{1}{2}/\frac{1}{2}$, tiene complejidad $O(n \lg n)$. ¿Qué pasa si parte el arreglo en proporción $\frac{1}{3}/\frac{2}{3}$? ¿Y $\frac{1}{k}/\frac{k-1}{k}$? (No hace falta que su prueba sea formal.)