Ing. de Sistemas e Informática
Estructuras de Datos y Algoritmos
Jorge Mario Londoño P.

Ejercicios de repaso 4

Métodos de Ordenación

1. Sea el siguiente arreglo de Chars:

M E T O D O S D E O R D E N A C I O N

a. Ilustrar la secuencia de pasos que se obtiene aplicando el algoritmo de ordenación por selección. Contabilizar el número de comparaciones e intercambios realizados.
b. Ilustrar la secuencia de pasos que se obtiene aplicando el algoritmo de ordenación por inserción. Contabilizar el número de comparaciones e intercambios realizados.
c. Ilustrar la secuencia de pasos que se obtiene aplicando el algoritmo de ordenación shellsort. Contabilizar el número de comparaciones e intercambios realizados.

2. *Natural Mergesort*

Una idea para organizar los elementos de un arreglo aprovechando que algunos de ellos pueden estar en orden, es buscar subrangos en orden y hacer la operación *merge* entre ellos. Más exactamente:

1. Incrementar un primer contador mientras se encuentre una secuencia creciente de elementos. El contador para en el primer elemento que cumpla que es menor que su predecesor.
2. A partir de la posición en donde termino el recorrido anterior, iniciar un segundo contador y hacer un recorrido identificando un segundo rango en orden.
3. A los dos rangos anteriores hacerles la operación merge. Luego del merge volver al paso 2 y repetir hasta organizar todo el vector.

Se pide entonces:

- Dar la implementación del algoritmo *Natural Mergesort*.
- Estimar el tiempo (# de comparaciones) en función del número de secuencias crecientes en el arreglo.
- Cuál sería el peor caso para este algoritmo. Cómo sería el tiempo en este caso.

3. *Aleatorización de una lista enlazada*

La estrategía “Divide y Vencerás” es frecuentemente utilizada para diseñar algoritmos. Como ejercicio para familiarizarse con esta estrategía se propone aleatorizar los elementos de una lista ordenada. Este tipo de operaciones son frecuentemente utilizadas en los programas (e.g. barajar un juego de cartas).

Se pide:

- Dar una implementación del metodo *shuffle* para listas enlazadas haciendo uso de la estrategía divide y vencerás.
- Estimar el tiempo (# de comparaciones, # de intercambios) requeridos por el algoritmo propuesto en función de la longitud de la lista N.
- (* Opcional) Mejorar el desempeño del algoritmo propuesto garantizando que el tiempo es linearitmético y el espacio adicional es logaritmico.

4. Algoritmos de ordenación no recursivos

Un principio fundamental de la programación establece que la recursión y la iteración son equivalentes: Cualquier programa recursivo se puede hacer iterativo o viceversa.

- (*) Dar una implementación iteractiva de mergesort
- (*) Dar una implementación iterativa de quicksort. Es la implementación dada un algoritmo *in-situ*?

5. Se tiene un arreglo genérico ordenado de menor a mayor. Convertir eficientemente el arreglo a orden mayor a menor. Reto adicional: Hacer el proceso *in-situ*.

6. Los algoritmos *naïve* para encontrar la intersección/diferencia de dos conjuntos son algoritmos $\sim N^2$. Si los conjuntos se encuentran en orden, es posible resolver estos problemas de forma más eficiente:

- Dar un algoritmo para encontrar la intersección de dos arreglos ordenados. Determinar el orden de crecimiento del algoritmo.
- Dar un algoritmo para encontrar la diferencia de dos arreglos ordenados. Determinar el orden de crecimiento del algoritmo.

7. Se desean comparar arreglos no ordenados para determinar si contienen los mismos elementos. El método *naïve* es $\sim N^2$. Dar una mejor solución partiendo de ordenar los arreglos a comparar, indicar el orden de crecimiento de su solución.

8. Encontrar los elementos duplicados en un arreglo. Indicar cuantas veces se repiten los elementos duplicados.

9. Dado un arreglo de entrada, determinar el número de inversiones entre sus elementos.

10. Dado uno de los valores a_i en un vector no ordenado, determinar que posición ocupa este valor en el ranking de elementos del vector. (El mayor elemento ocupa la posición 1 del ranking y el menor la N).

12. Se dice que dos palabras son anagramas si contienen exactamente las mismas letras (en igual cantidad), pero en distinto orden. Por ejemplo son anagramas “altisonancia” y “nacionalista”. Dar un algoritmo que reciba dos String y determine si son anagramas.

11. Distribución de probabilidad acumulada muestral

Se tienen un conjunto de datos $[x_0, \ldots, x_{n-1}]$ no ordenado.

Se define la distribución de probabilidad acumulado $P(x_j<Kte)$ como la probabilidad de que uno de los datos sea menor a una constante Kte. Observando que esta probabilidad se corresponde con el porcentaje de datos que son menores a la Kte, sería sencillo determinar la probabilidad acumulada una vez se ordenen los datos.

Diseñar un ADT ProbabilidadAcumulada que se inicialice con un arreglo de muestras y que calcule la probabilidad acumulada para valores de entrada x_j suministrados como parámetro del método.

Algoritmos recursivos y Recurrencias

1. El siguiente es una versión recursiva para el cálculo del factorial:

```
public class Factorial {

	public static long factorial(long n) {
		if (n >= 0) {
			if (n <= 1)
				return n;
			else
				return n * factorial(n - 1);
		} else
			throw new IllegalArgumentException("En factorial se requiere n>=0");
	}

	public static void main(String[] args) {
		StdOut.println("factorial(" + 5 + ") = " + factorial(5));
		StdOut.println("factorial(" + 20 + ") = " + factorial(20));
		StdOut.println("factorial(" + 70 + ") = " + factorial(70)); // Es correcto? Qué
ocurre?
	}

}
```

a) Plantear la recurrencia que describe el número total de multiplicaciones.

b) Dar solución a esta recurrencia.

2. El siguiente algoritmo resuelve el problema de las “Torres de Hanoi”

```
public class Hanoi {

    public static void hanoi(int disks, int source, int dest, int aux) {
        if (disks == 1) {
            StdOut.println("Move 1 disk from " + source + " to " + dest);
        } else {
            hanoi(disks - 1, source, aux, dest);
            StdOut.println("Move 1 disk from " + source + " to " + dest);
            hanoi(disks - 1, aux, dest, source);
        }
    }

    public static void main(String[] args) {
        hanoi(3, 1, 3, 2);
    }

}
```

a) Plantear la recurrencia que describe el número total de movimientos.
b) Dar solución a esta recurrencia.

3. La siguiente recurrencia define los números de Fibonacci:

$$fib(n)=\begin{cases} n\ , & n\leqslant 1 \\ fib(n-1)+fib(n-2)\ , & n\geqslant 2 \end{cases}$$

a) Dar el pseudo-código del algoritmo recursivo.

b) Determinar el número de sumas necesarias para calcular fib(n).

4. La potenciación con potencia entera utilizando la definición requiere (n-1) multiplicaciones:

$$x^n = x \cdot x \ldots x$$

Una mejor forma de calcular potencias enteras de x es utilizando la recurrencia:

$$x^n=\begin{cases} x^{n/2}\cdot x^{n/2}\ , & si\ \ n\ \ es\ \ par \\ x\cdot x^{n/2}\cdot x^{n/2}\ , & si\ \ n\ \ es\ \ impar \end{cases}$$

(Observar que $x^{n/2}$ se debe calcular una sola vez y que n/2 representa la división entera por 2).

a) Dar el pseudo-código de este algoritmo.

b) Estimar el número de multiplicaciones en función de n.