Departamento de Computación
FCEFQyN, Universidad Nacional de Río Cuarto
Asignaturas: Diseño de Algoritmos - Algoritmos II
Primer Cuatrimestre de 2021

Guía Práctica No. 1: Revisión, Fuerza Bruta

Esta guía práctica abarca tanto revisión de conceptos previos como una de las técnicas algorítmicas más básicas, Fuerza Bruta. Refiere por lo tanto a los capítulos 1, 2 y 3 de *Introduction to the Design and Analysis of Algorithms* (Levitin 2003), que son lectura recomendada para el desarrollo de la práctica.

Parte de esta práctica cuenta con esquemas de programación y tests de asistencia a su resolución. Éstos pueden accederse a partir del classroom de github disenho-de-algoritmos-unrc-2021, para el cual es conveniente tener una cuenta en Github (`http://github.com`).

Esta práctica no tiene entrega formal. Su resolución es opcional.

1. Para cada uno de los siguientes algoritmos, indique qué problema resuelve y cuál es su tiempo de ejecución en el peor caso.
   - Mergesort — ordena una secuencia de elementos, y su orden en peor caso es $O(n \log n)$.
   - Quicksort
   - Algoritmo de Dijkstra
   - Algoritmo de Warshall
   - Algoritmo de Kruskal
   - Búsqueda binaria
   - Algoritmo de Euclides
   - Búsqueda a lo ancho
2. Para cada uno de los siguientes problemas, indique cuán eficientemente pueden resolverse:
   - Ordenar una secuencia — Puede resolverse en $O(n \log n)$
   - Buscar un elemento en un conjunto
   - Problema de la exponenciación (calcular a^n)
   - Eliminación Gaussiana
   - Encontrar par de elementos más cercanos en un conjunto de puntos del plano
   - Composición relacional
   - Ordenamiento topológico
   - Determinar existencia de ciclos eulerianos en un grafo
3. Considere el siguiente algoritmo:

```
int powerOfTwo(int n) {
    if (n == 0) return 1;
    return (powerOfTwo(n-1) + powerOfTwo(n-1));
}
```

   Demuestre que este programa, en el peor caso, tiene tiempo de ejecución de orden $O(2^n)$ (realiza una cantidad exponencial de sumas).

4. Considere el siguiente algoritmo:

```
insertionSort(int[] array) {
    //inv: sorted(array, 0, j-1)
    for (j = 1; j < array.length; j++) {
        int key = array[j];
        int i = j - 1;
        //inv: sorted(array, 0, j-1) && ...
        while (i >= 0 && array[i] > key) {
            array[i + 1] = array[i];
            i = i - 1;
        }
        array[i + 1] = key;
    }
}
```

   Proponga un fortalecimiento para el invariante del ciclo más interno, y demuestre que el cuerpo de ciclo lo preserva.

5. Considere la clase `ArraySorter` provista en el repositorio. Implemente los algoritmos de ordenamiento `insertionSort`, `bubbleSort` y `mergeSort`. Asegúrese que todos los tests pasen (debe completar la implementación de algunos métodos auxiliares). Complete el conjunto de tests de unidad provistos, con otros adicionales, para los algoritmos implementados. Utilice el proyecto para realizar una comparación experimental del tiempo de ejecución de los algoritmos implementados.

6. Una forma de implementar el algoritmo de ordenamiento *heapsort* es mediante el uso de una implementación eficiente de colas de prioridades: los elementos de la secuencia a ordenar se trasladan a la cola de prioridades, y luego se construye incrementalmente una secuencia ordenada mediante la extracción (ordenada) de elementos de la cola de prioridades. Implemente *heapsort* en Java, siguiendo este proceso, incorporándolo a la clase `ArraySorter`. Complete su implementación con tests de unidad y las directivas necesarias para que forme parte de la comparación experimental del tiempo de ejecución de todos los algoritmos de ordenamiento, incluyendo *heapsort*.

7. Realice experimentos para comparar los tiempos de ejecución de los tres algoritmos para computar el máximo común divisor presentados en la sección 1.1 de "Introduction to the Design and Analysis of Algorithms" (Levitin 2003). Utilice las clases provistas para este fin, completando, además de las implementaciones de los algoritmos, casos de tests en las clases correspondientes.

8. Considere el siguiente algoritmo, que analiza si hay elementos duplicados en un arreglo.

```
var hasDuplicates = function(arr) {
  for (var i = 0; i < arr.length; i++) {
    var elem = arr[i];
    if (arr.slice(i + 1).indexOf(elem) !== -1) {
      return true;
    }
  }
  return false;
};
```

   Argumente rigurosamente, sin necesariamente realizar un análisis detallado de tiempo de ejecución, que este algoritmo es $\Theta(n^2)$

9. Para los siguientes problemas, describa en palabras y/o pseudo-código, algoritmos que los resuelvan basándose directamente en la descripción del problema:

   - Decidir si un conjunto de enteros se puede particionar en dos conjuntos de igual suma.

- Dadas dos cadenas, decidir si las mismas son anagramas (una es permutación de la otra).
- Dado un número natural n, descomponerlo en sus factores primos.
- Dadas dos cadenas p y s, decida si p es subcadena de s.
- Dadas dos cadenas p y s, decida si p es subsecuencia de s (los elementos no necesariamente tienen que aparecer contiguos en s).
- Dada una secuencia s de números, encontrar el k-ésimo elemento más grande en s.

10. Escriba programas en Java y Haskell que calculen toda las permutaciones de una lista, todos los subconjuntos de un conjunto, y todas las sublistas de una lista. Utilizando estas rutinas, resuelva los siguientes problemas:

    - Decidir si dos secuencias son anagramas.
    - Dado un conjunto s y un valor n, decidir si existe un subconjunto de s cuya suma sea n.
    - Dadas dos cadenas p y s, decida si p es subcadena de s.
    - Dadas dos cadenas p y s, decida si p es subsecuencia de s (los elementos no necesariamente tienen que aparecer contiguos en s).

11. Escriba un programa Haskell que implemente el algoritmo de ordenamiento *SlowSort*, que ordena una lista de elementos (por ejemplo, de enteros) mediante la generación de la lista de todas las permutaciones de la lista original, y filtrando aquella que está ordenada. Complete el entorno de algoritmos de ordenamiento para Java con una implementación de *slowSort*.

12. Diseñe, e implemente en Java o Haskell, un algoritmo que elimine elementos repetidos de una lista. Argumente sobre la complejidad en peor caso de su algoritmo.

13. Considere la clase `CaesarCracker`, provista en el repositorio, y cuyo propósito es desencriptar, mediante fuerza bruta, un mensaje ASCII cifrado mediante un cifrado César. Implemente el método `decode`, que dado un mensaje y la clave, desencripta el mismo.

14. Considere nuevamente la clase `CaesarCracker`. Implemente el método `bruteForceDecrypt()`, que intenta desencriptar, por fuerza bruta, un mensaje encriptado, probando con todas las claves posibles de hasta tamaño k (véase atributo `passwordLength`) y utilizando como criterio de desencripción exitosa que el mensaje desencriptado contenga una palabra específica (véase atributo `messageWord`). Argumente sobre la complejidad de su solución.