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

Guía Práctica No. 2: Divide & Conquer / Decrease & Conquer

Esta guía práctica abarca las técnicas de diseño Decrease & Conquer y Divide & Conquer. Refiere por lo tanto a los capítulos 4 y 5 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.

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

1. Dado un arreglo `A[0..n-1]` de números reales distintos, llamaremos *inversión* a un par de valores `(A[i],A[j])` almacenados en el arreglo tales que `A[i]` $>$ `A[j]`, con `i` $<$ `j`. Diseñe, utilizando Divide & Conquer, un algoritmo que calcule el número de inversiones en un arreglo dado. Calcule el tiempo de ejecución en peor caso de su algoritmo, e impleméntelo en Java.

2. Aplicando el Teorema Maestro, decida cuál es la tasa de crecimiento de las siguientes ecuaciones de recurrencia:

   - $T(1) = 1; T(n) = 4T(\frac{n}{2}) + n$
   - $T(1) = 1; T(n) = 4T(\frac{n}{2}) + n^2$
   - $T(1) = 1; T(n) = T(\frac{n}{2}) + 1$
   - $T(1) = 1; T(n) = 2T(n-1) + 2n$
   - $T(1) = 1; T(n) = T(\frac{n}{2}) + 2^n$
   - $T(1) = 1; T(n) = 2T(\frac{n}{2}) + \log n$

3. Una solución Divide & Conquer (D&C) a un problema determinado consta de:

   - un criterio *isBase* de distinción para las entradas, que separe los casos base y los casos complejos,
   - un proceso directo *base* de resolución de los casos base,
   - un proceso *split* que, dada una entrada compleja (no base), descomponga a la misma en un número de entradas más simples,
   - un proceso *join* que, dadas las soluciones a las entradas más simples creadas por split, construya una solución al problema original.

   Dados estos elementos, se puede construir un algoritmo D&C para resolver el problema. En Haskell, se puede sintetizar este proceso mediante una función de alto orden, que tome por entradas todos los elementos anteriores que caracterizan una solución D&C, y los combine para conseguir implementar la función correspondiente:

```
evalDC::Eq a⇒(a→Bool)→(a → b)→(a→[a])→([b]→b)→a→b
evalDC isBase base split join x | isBase x = base x
                                | otherwise = join (map ( evalDC isBase base split join) (split x ))
```

   Usando esta función, se pueden construir soluciones D&C simplemente identificando e implementando adecuadamente *isBase*, *base*, *split* y *join*. Por ejemplo, la función que computa números de Fibonacci puede definirse usando *evalDC* de la siguiente manera:

```
fibonacci = evalDC (\x→x<=1) (\x→x) (\x→[x-1,x-2]) (\l→(head l)+(last l))
```

Considere el problema de, dada un par de secuencias p y t, determinar si p es subsecuencia (de elementos contiguos) de t. Identifique las funciones isBase, base, split y join para resolver este problema mediante D&C, y utilice evalDC para conseguir el programa que resuelve el problema.

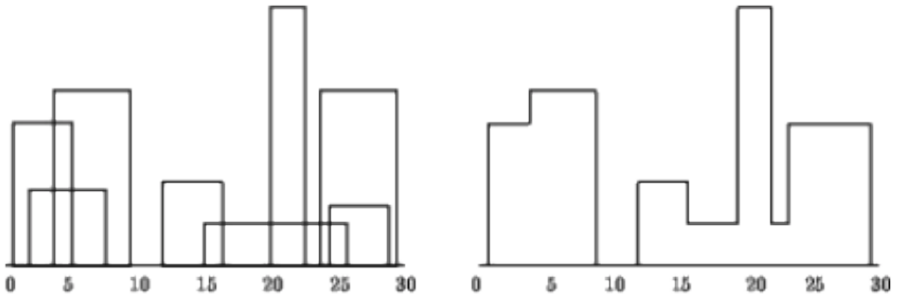
4. Explique cuál es la relación entre *inducción* y *recursión*. Si le parece que hay una correspondencia entre estos conceptos, a qué le parece que corresponde la *hipótesis* inductiva en la definición de una función recursiva?

5. Implemente en Java o Haskell el algoritmo para resolver el problema de, dado un conjunto de puntos del plano, encontrar el par de puntos (distintos) cuya distancia es la menor entre todos los puntos del plano, en $O(n \log n)$, discutido en *Introduction to the Design and Analysis of Algorithms* (Levitin). Explique además las razones por las cuales no es necesario comparar *todos* los pares de puntos en las inmediaciones de la línea media, sino que para cada punto a uno de los lados de esta línea, sólo hace falta considerar a lo sumo 6 puntos del otro lado.

6. Diseñe usando Divide & Conquer e implemente en Java un algoritmo para resolver el problema de encontrar la subsecuencia de suma mínima, de una secuencia dada. Realice el análisis de tiempo de ejecución para el peor caso de su algoritmo. En caso de tener complejidad superior a $O(n \times \log n)$, optimice su algoritmo para conseguir tal eficiencia.

7. Dado un árbol binario de números enteros, se desea calcular la máxima suma de los nodos pertenecientes a un camino entre dos nodos cualesquiera del árbol. Un camino entre dos nodos n_1 y n_2 está formado por todos los nodos que hay que atravesar en el árbol para llegar desde n_1 hasta n_2, incluyéndolos a ambos. Diseñe usando Divide & Conquer e implemente en Java o en haskell un algoritmo para resolver este problema. Puede suponer que el árbol está balanceado. El algoritmo debe tener una complejidad temporal de peor caso igual o mejor que $O(n \times \log n)$ siendo n la cantidad de nodos del árbol.

8. Diseñe usando Divide & Conquer e implemente en Java un algoritmo para resolver el problema de dibujar la silueta de los edificios de una ciudad (skyline). Dada la ubicación y forma de n edificios rectangulares en 2-dimensiones, computar la silueta de los edificios eliminando las líneas superpuestas.



9. Implemente en Java soluciones Decrease & Conquer con decremento por una constante y por un factor constante para el problema de multiplicar dos números enteros. Calcule además el número de sumas efectuadas por su algoritmo, en función del tamaño de la entrada.

   En relación a una de sus soluciones, considera que es más económico computar $n + n$ o $2 \times n$? Justifique su respuesta.

10. Utilizando la técnica Decrease & Conquer, diseñe un algoritmo para encontrar los elementos mayor y menor de una secuencia de n enteros positivos. Implemente su algoritmo en Haskell.

11. Utilizando la técnica Decrease & Conquer, diseñe un algoritmo para encontrar el índice del mayor elemento de una secuencia de n enteros positivos. Piense en una alternativa a este algoritmo diseñado utilizando Fuerza Bruta, y compare implementaciones para estos dos algoritmos en Haskell.

    Realice además el análisis correspondiente para calcular cuántas comparaciones de elementos son realizadas por ambos algoritmos en el peor caso.

12. Diseñe usando Decrease & Conquer e implemente en Java o Haskell un programa que, dada una lista de asignaturas de un plan de estudios y un conjunto de correlatividades de cursado (cada una de las cuales indica un pre-requisito para el cursado de una materia), determine si el plan es *consistente*, es decir, si el mismo contiene correlatividades cíclicas. Su algoritmo no sólo debe emitir el *veredicto* (consistente o inconsistente), sino que en caso de consistencia debe dar un ejemplo, es decir una forma de cursar asignaturas en cuatrimestres consecutivos de manera tal de respetar las correlatividades.

13. Diseñe usando Divide & Conquer e implemente en Java y Haskell un programa que, dadas dos secuencias de caracteres, construya la subsecuencia común a ambas de longitud máxima. Se entiende por *subsecuencia* una cadena de caracteres que se deriva de la original mediante la eliminación de caracteres pero sin cambiar el orden de los caracteres en la secuencia original.