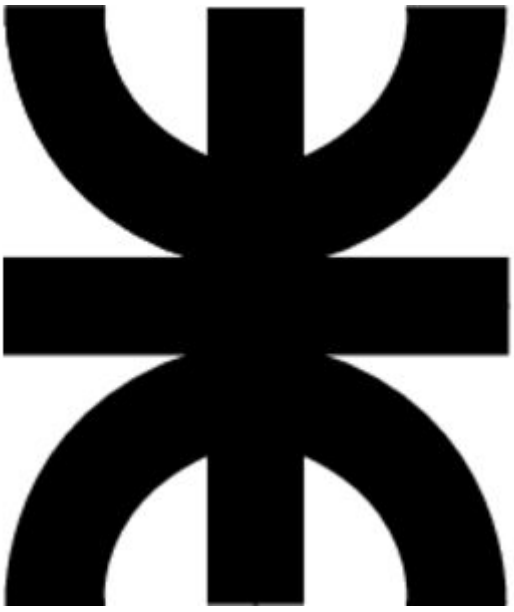

UTN – Facultad Regional Resistencia

TRABAJO PRÁCTICO INTEGRADOR DE TÉCNICAS DE DISEÑO Y DESARROLLO DE ALGORITMOS

GRUPO N°: 5
INTEGRANTES: BRITOS, Miguel
GASPARUTTI, Edgardo
IMFELD, Facundo
OJEDA NUÑEZ, Pablo
RETAMOZO, Agustín
PROFESORES: TORTOSA, Nicolás – ACUÑA César
FECHA: 25/06/2019

Problema N° 1:
Realizar un algoritmo eficiente que dado los rangos 1000, 10000, 100000 halle los números amigos y los retorne en forma de tupla.
Estructura y Funcionamiento:
En el ejercicio planteado definimos dos funciones: divisores y amigos.
Divisores recibe como parámetro un número donde su objetivo es retornar la sumatoria de todos los divisores del número que recibió como parámetro.
Amigo es invocada como función principal y recibe como parámetro el rango máximo del conjunto de números que se quiere buscar los pares de números amigos y a su vez, ésta función va invocando a **divisores** cada vez que requiera hallar la sumatoria de los divisores de cada número a medida va analizando los números del conjunto.
Técnica utilizada: **Divide y Vencerás**

- La ventaja que nos dio esta técnica fue la posibilidad de verificar en funciones distintas los dos problemas principales que tenía el ejercicio, la de hallar los divisores del número en cuestión y la comprobación de las parejas de las sumas de divisores.

Problema N°2:
Se cuenta con información previamente cargada (en una lista, diccionario, archivo, etc) de:
a) Sucursales: id, nombre.
b) artículos: id, descripción, precio
c) Articulos_x_Sucursal: id, id_articulo, id_sucursal, cantidad
Se necesita un informe que enumere los artículos cuya disponibilidad en una o varias sucursales sea cero, y en las otras mayor a 5 (al menos en una).

Estructura y Funcionamiento:
1_tabla.py genera aleatoriamente los datos con los que se van a trabajar en los archivos artículos, artxsucursal y sucursal pudiéndose elegir la cantidad de productos con los que se quiere trabajar y cuánto de ellos se requiere cumpla la condición para después verificar posteriormente el resultado.
2_exportar_tabla.py copia los datos los archivos de articulos, artxsucursal y sucursal para verificar que los productos fueron listados correctamente.
3_trabajo_2.py consiste en 3 funciones:

- **buscar stock cero** que recibe como parámetro el archivo artículos por sucursal y retorna todos aquellos artículos que tienen como stock cero en al menos 1 sucursal y si se encuentra el mismo artículo en otra sucursal, este no se repite.
- **buscar stock mayor que cinco** recibe el resultado de todos los productos que tienen como stock cero y vuelve a recorrer el mismo archivo y pregunta si alguno de esos tiene cantidad mayor a 5.
- **sort** ordenamos la lista por id de producto

- **listar productos** recibe como parámetro una lista que contiene la lista generada anteriormente con productos cantidad = 0 y cantidad>5 y los muestra por pantalla.

Técnica utilizada: **divide y vencerás**.

- Esta técnica la aplicamos porque nos permitió poder encontrar de forma separada los productos que cumplen con la condición de tener stock igual a cero y de los que tienen mayor que 5 reduciendo de esa manera a 2 la cantidad de recorridos al archivo de artículos por sucursal obteniendo una complejidad lineal.

Problema N°3:

Realizar un algoritmo que dado una jugada elija la mejor posición de juego en un tablero de ta-te-tí. Se evaluará el tiempo de respuesta del algoritmo ante una jugada.

Estructura y Funcionamiento:

La función principal se compone de un ciclo de iteración que se ejecuta mientras existan celdas vacías y no exista un ganador.

- **juegaHumano**: Inicia jugando una persona seleccionando una de las 9 celdas disponibles, la disponibilidad de celdas es controlada por una función tomarMovimiento y se verifica si existe un ganador.
- **tomarMovimiento** que devuelve un booleano, en caso correcto la función almacena la ficha del jugador en la celda.
- **juegaPC**: primero calcula la cantidad de celdas vacías con la función celdas_vacias para estimar la profundidad que va a tener el árbol de decisión, luego aplica la función minmax para determinar la mejor opción disponible.
- **minmax**: esta función recibe como parámetros el estado actual del tablero, la profundidad actual del árbol y el jugador de turno; inicializa una variable mejor que contiene una coordenada nula y HV dependiendo si se trata de la persona o la máquina, luego analiza si existe un ganador o si ya no se puede expandir mas el árbol, en base a eso evalúa el estado actual y retorna si hubo un empate o el valor del ganador (+1/-1) si esto no sucede la función continua analizando las celdas vacías haciendo llamadas recursivas de la función minmax para determinar la coordenada x,y de la celda de mejor opción, cuando finaliza la recursión verifica si el valor obtenido es la mejor opción para continuar.

Técnica Utilizada: **Backtracking**

- Utilizamos backtracking porque era la única técnica que nos garantizaba analizar todas las posibles jugadas por el oponente y así elegir la mejor posición que nos garantice una victoria o un empate en el peor de los casos.

Problema N°4:
Elegir una problemática o un juego a desarrollar en python aplicando alguna de las técnicas aprendidas durante el módulo de Complejidad.

Para este item decidimos elaborar un programa capaz de resolver un tablero sudoku de 9x9 donde tendremos que indicarle al programa cuál es el tablero que deseamos lo resuelva por nosotros.

Estructura y Funcionamiento:

Nuestro algoritmo consta de 3 funciones requerida para llevar el análisis de la solución y una función **renderizado** que se encarga de dibujar una tabla en la consola, utilizando caracteres especiales para simularla.

- **Buscar:** función que recibe como parámetro el tablero a resolver y nos retorna la posición (índices) en donde tengamos el primero 0 (el número 0 representa una celda que debemos completar con algún número del 1 al 9).
- **Solucion:** es nuestra función solución que recibe como parametro el tablero, el numero que queremos escribir y en dónde los queremos escribir (posicion). A través de procesos iterativos analizamos tanto filas como columnas de todo el tablero dividido en 9 cuadrantes de 3x3 que conforman al tablero completo de 9x9. Si llega a encontrar en algún cuadrante… en alguna fila… en alguna columna que existe un número igual al que nos pasan por parámetro, inmediatamente esta función nos retorna FALSE y así verificamos que no va a poder ser posible escribir ese número en esa celda. En caso contrario, todo está OK y la función nos retorna TRUE.
- **Soduku:** es la función principal que nos solicita le demos el tablero que deseamos resolver. Lo primero que hace esta función es llamar a la función **buscar** para obtener la primera “celda” que debemos completar indicándonos posiciones por valores x e y que formarán parte del índice de la matriz **board** que pasamos por parámetro. Después vamos probando combinaciones del 1 al 9 en ese mismo orden mientras que al mismo tiempo por cada número que probamos llamamos a la función **solución** que nos retornará si es o no un número válido, en caso de no serlo avanzará al siguiente número hasta encontrar una solución válida. Una vez hayamos encontrado una solución para esa celda, escribimos, y haremos la recursión llamando a **sudoku** nuevamente y pasándole el tablero actualizado con la nueva solución y repitiendo el proceso. Si llega a darse el caso de que nos encontremos con que un casillero no tiene solución después de haber probado los 9 números disponibles, haremos el backtracking y probaremos una nueva combinación para la casilla anterior y así recursivamente hasta poder rellenar todo el tablero.

Técnica Utilizada: **Backtracking**

La mejor técnica que nos garantizará una solución en caso de que el tablero tenga una, de no ser así, el ejercicio no resuelve y obtendremos como resultado el mismo tablero tal como se lo pasamos.

Conclusión:
Al finalizar el trabajo realizamos una comparación entre las técnicas que utilizamos, la técnica de divide y vencerás nos pareció la más sencilla de utilizar por su facilidad de implementación y comprensión, pero nos encontramos con la dificultad de que los tiempos de ejecución se disparaban a medida que los datos de entrada se incrementaban, esto se da por los órdenes de complejidad cuadrática en el caso caso del problema N° 1 y lineal en el problema N° 2.
Sin embargo al buscar soluciones óptimas en los problemas N°3 y N°4 nos vimos obligados a utilizar backtracking por la necesidad de recorrer todas las combinaciones posibles para hallar la solución óptima.