CENTRO DE CIENCIAS BÁSICAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

ACADEMIA DE INDUSTRIA DEL SOFTWARE

Nombre del Estudiante:	José Luis Sandoval Pérez	**Fecha:**	28/02/2024
Materia:	**TEORIA DE LA COMPLEJIDAD COMPUTACIONAL**	**Carrera:**	ICI
Profesor:	Dr. Miguel Angel Meza de Luna	**Semestre:**	6º
Periodo:	(X) Enero – Junio ( ) Agosto - Diciembre	**Aciertos:**	
Tipo de Examen:	Parcial: 1° (X) 2° ( ) 3° ( ) Otro:	**Calificación:**	

Instrucciones: Lee el código de honor.

Código de Honor

Prometo seguir el código de honor y obedecer las reglas para tomar este examen:

- Trabajaré de manera individual en este examen, todas las respuestas serán de mi propio intelecto.
- No compartiré mis respuestas del examen con alguien mas.
- No emplearé una identidad falsa, ni tomaré este examen en nombre de alguien mas.
- No tomaré parte en actividades deshonestas para mejorar mi resultado o influenciar en el de alguien mas.

Acepto

I. Instrucciones: Contesta correctamente lo solicitado.	**Valor: 6 pts.**

1. Una característica que permiten medir la eficiencia de un algoritmo es en función de la cantidad de memoria utilizada, explica cómo hacer la medición de memoria utilizada por un programa.
   Se puede medir en base a las operaciones que se hacen y la cantidad de memoria que se utiliza para cada operacion
2. ¿Por qué T(n) no se expresa en tiempo de ejecución?
   Debido a que este tiempo de ejecución puede variar dependiendo de el hardware de la maquina en la que se este ejecutando, podemos decir que para una misma entrada el tiempo será diferente para 2 máquinas distintas.
3. ¿Cuál es el timpo de ejecución de la función factorial? Explica como llegaste al resultado.
   El tiempo de ejecución la función factorial es t(n)=n-1, la respuesta es bastante sencilla el algoritmo recursivo se ejecuta n-1 veces, ya que existe una condición de paro que es caso base que cuando n=1 el resultado del factorial es 1. Por lo tanto ese caso base no se toma en cuenta y se dice que el tiempo es t(n)=n-1
4. Para buscar un elemento de un vector cuál es el T(n) para el mejor caso, cuál es el T(n) para el peor caso y cuál es el T(n) para el caso promedio. También indica por qué el T(n) = n en este problema.
   El t(n) para el mejor caso seria t(n)=1 debido a que puede ser que en el primero elemento del vector este el numero que se desea buscar. El peor de los casos sería cuando el elemento a buscar no este en el vector, es decir se tienen que recorrer todos los elementos del vector aunque no se encuentre y el t(n)=n. En un caso promedio lo definiría como t(n)=log n, debido a que la búsqueda no es completa, sino la búsqueda se pudiera dividir en segmentos pequeños si

se encuentra antes de recorrer todo el arreglo. Y t(n) de este problema es 'n' debido a que en una búsqueda lo más común es que se recorran los arreglos debido a que la mayoría de las veces nos dan un vector que no esta ordenado y la búsqueda se vuelve mas extensa. Pudo decir que también considero que el t(n) promedio pudiera ser 'n' por la misma razón anterior.

5. Explica por qué el Big O de ***3 x n + log(n) +1*** es ***O(n)***
   Sabemos que en la notación big O siempre se mide en base a el peor de los casos, es decir la que mas crecimiento genera y es mas ineficiente, en la función anterior podemos darnos cuenta que existen 2 crecimientos constantes los cuales son '3xn' y '1'. También notamos que tenemos log(n), como mencione big O siempre será el caso peor entonces podemos eliminar las constantes y los logaritmos, ya que no presentan un crecimiento significativo en la complejidad, y nos queda únicamente '3n', de igual manera podemos eliminar la constante que lo acompaña por la misma razón, es por eso que podemos decir que el big O es O(n).
6. Explica qué es n_0, recuerda que este concepto se vio cuando se grafica el comportamiento del algoritmo y alguna notación asintótica
   n0 hace alusión al punto en donde se cumple un limite para cualquier de las notaciones y el comportamiento que tiene el algoritmo.
7. **Explica por qué Algunas soluciones recursivas repiten cálculos innecesariamente**
   Porque pueden existir re declaraciones de variables que la función inicializa desde un principio, puede existir llamadas innecesaria debido a que posiblemente ya se haya llegado a la solución pero aun así se ejecute, es por eso que son más ineficientes que los algoritmos iterativos
8. Explica la relación entre NPC y Big O
   Existe una relación poco notoria, el análisis de problemas NPC se enfoca en el problema en si, si es que puede ser resuelto o no, mientras que Big O se basa en la solución. Ambos problemas tienen una relación debido a que los problemas NPC son difícil de resolver y comprobar y el big O es el caso en donde la solución se resuelve en el peor de los casos. Es la relación que yo veo.
9. Mediante un ejemplo explica el tipo de problema decidible intratable
   Empezamos diciendo que los problemas decidibles e intratables son los problemas en los cuales existe un algoritmo que resuelve el problema pero no en un tiempo considerable. Un ejemplo podría ser el de los números perfectos, debido a que existe un algoritmo que puede resolverlo, pero el tiempo en que obtiene los números perfectos no es considerablemente aceptable.
10. Explica con el ejemplo de rompecabezas el concepto de NP y con el ejemplo del sudoku el concepto de NPC.
    El concepto NP básicamente es que existe una solución que es difícil de encontrar, pero es fácilmente comprobable, esto se adapta a un rompecabezas, es un poco complicado ver que pieza encaja con cual pero al comprobarlo es fácil debido a que notamos que las piezas unidas forman la imagen correcta.
    A diferencia de los problemas NPC que son problemas que son difíciles de resolver y difíciles de comprobar, la explicación para el suduku es bastante similar a la del rompecabezas sin embargo es difícil resolver un suduku debido a que debemos de estar siempre pendientes a los números que hay en la casilla y no debemos de cometer errores. Y la manera en la que se verifica la solución de igual manera es bastante difícil debido a todos los cálculos que conlleva hacerlo.

II. Determina el Big O de las siguientes funciones recursivas y justifica como llegaste al resultado	**Valor: 2 pts.**

1. Función 1

```
int Recursiva1 (int n) {

  if (n <= 1)
     return(1);
  else
     return(Recursiva1(n-1)+ Recursiva1(n-1));
}
```

El big O de esta función es igual a O(n) la justificación es la siguiente, cada llamada recursiva obtiene otras 2 llamadas a la función es decir 2 a la n potencia, y este genera una estructura

similar a la de un árbol, esto quiere decir que mientras el valor de n aumenta también las distintas llamadas a la función y el árbol se hace mas grande es por eso que el Big O es O(2^n)

2. Función 2

```
int Recursiva2 (int n) {

  if (n <= 1)
     return (1);
  else
     return(2 * Recursiva2(n-1));
}
```

Podemos darnos cuenta que la función recursiva solo se llama una vez en cada llamada recursiva a pesar de que se este multiplicando por una constante. Por lo tanto el big O es O(n)

3. Función 3

```
int Recursiva3 (int n) {

  if (n <= 1)
     return (1);
  else
     return (2 * Recursiva3(n /2));
}
```

Notamos que en esta función recursiva en cada llamada de la función el numero de n se divide en 2, es decir, ya no tiene complejidad O(n), al ser una division y al reducirse el numero de llamadas que hará la función podemos decir que la complejidad big O de este algoritmo es O(log n)

4. Función 4

```
int Recursiva4 (int n, int x) {
   int i;
   if (n <= 1)
        return (1);
   else {
            for (i=1;i<=n;i++) {
                x = x + 1;
             }
             return( Recursiva4(n-1, x));
     }
}
```

Notamos que cuando n>1 el ciclo for nos da una notación O(n) después esta la llamada recursiva que de igual manera por el código nos da una notación O(n) en el ciclo for esto quiere decir que por cada valor de n existen 2 O(n) por lo tanto podemos decir que el big O es O(n^2)

III. Instrucciones: Después de realizar el apartado anterior obtén el Big O de cada función según la librería de Python y compara tu resultado	**Valor: 2 pts.**

```
1
2
4
8
16
32
64
128
256
Exponential: time = 2.2E-06 * 1.9^n (sec)
```

1. La ejecucion usando la librería python sí coincidio con mi resultado.

```python
import big_o ##importamos libreria que calcula big o

def recursiva1(n):
    if n <= 1:
        return 1
    else:
        return recursiva1(n-1) + recursiva1 (n-1);

for x in range(10):
  print(recursiva1(x))


print(big_o.big_o(recursiva1, big_o.datagen.n_, n_repeats=10, min_n=1, max_n=2)[0])
```

2.
```
Constant: time = 7.9E-05 (sec)
```
La ejecucion utilizando la librería en python si coincidio con mi resultado.

```python
import big_o ##importamos libreria que calcula big o

def recursiva1(n):
    if n <= 1:
        return 1
    else:
        return 2*recursiva1(n-1);



print(big_o.big_o(recursiva1, big_o.datagen.n_, n_repeats=10, min_n=1, max_n=100)[0])
```

3.
```
1
1
2
4
4
8
8
8
8
16
Constant: time = 7.3E-06 (sec)
```
La ejecucion con la librería en python no coinicidio con mi resultado.

```python
import big_o ##importamos libreria que calcula big o

def recursiva1(n):
    if n <= 1:
        return 1
    else:
        return 2* recursiva1(n/2)

for x in range(10):
  print(recursiva1(x))


print(big_o.big_o(recursiva1, big_o.datagen.n_, n_repeats=10, min_n=1, max_n=10)[0])
```

4.
```
Constant: time = 2.4E-05 (sec)
```
La ejecucion con la librería dentro de python no coinicidio con mi respuesta.

```python
import big_o ##importamos libreria que calcula big o

def recursiva1(n):
    if n <= 1:
        return 1
    else:
        for x in range(n):
            x=x+1
    return(recursiva1(n-1))



print(big_o.big_o(recursiva1, big_o.datagen.n_, n_repeats=10, min_n=1, max_n=10)[0])
```