ESCUELA POLITÉCNICA NACIONAL

ESCUELA DE FORMACIÓN DE TECNÓLOGOS

ALGORITMOS Y ESTRUCTURAS DE DATOS (TSDS)

ASIGNATURA:	ALGORITMOS Y ESTRUCTURAS DE DATOS
PROFESOR:	Ing. Lorena Chulde MSc.
PERÍODO ACADÉMICO:	2023-B

TAREA 5

TÍTULO:

DISEÑO DE ALGORITMOS ESTRUCTURAS ITERATIVAS O

Nombre del estudiante: Josué Guerra, Carlos Pérez, Richard Soria

2023-B

PROPÓSITO DE LA TAREA

Aplicar sentencias de algoritmos mediante las estructuras de cíclicas WHILE, DO-WHILE, FOR para la resolución de ejercicios sencillos.

Realizar la investigación sobre:

DISEÑO DE ALGORITMOS: Recursividad Análisis y complejidad de un algoritmo

Las indicaciones del formato adjunto el siguiente link:

Indicaciones:

a. Revisa el material facilitado en la clase.

b. Formar grupos de 3 estudiantes, realizar la consulta indicada en word y realizar la presentación para:

c. Exposición de los contenidos fundamentales sobre recursividad y las reglas básicas para trabajar con recursividad y ponerlo en práctica a través de ejercicios.

d. Exposición de los contenidos fundamentales sobre el análisis, complejidad de un algoritmo y las reglas básicas para determinar el análisis y la complejidad de un algoritmo y ponerlo en práctica a través de ejercicios.

CONSULTA

CONTENIDOS FUNDAMENTALES SOBRE RECURSIVIDAD Y LAS REGLAS BÁSICAS PARA TRABAJAR CON RECURSIVIDAD

CONCEPTOS

Es una alternativa diferente para implementar estructuras de repetición (ciclos). Los módulos se hacen llamadas recursivas.

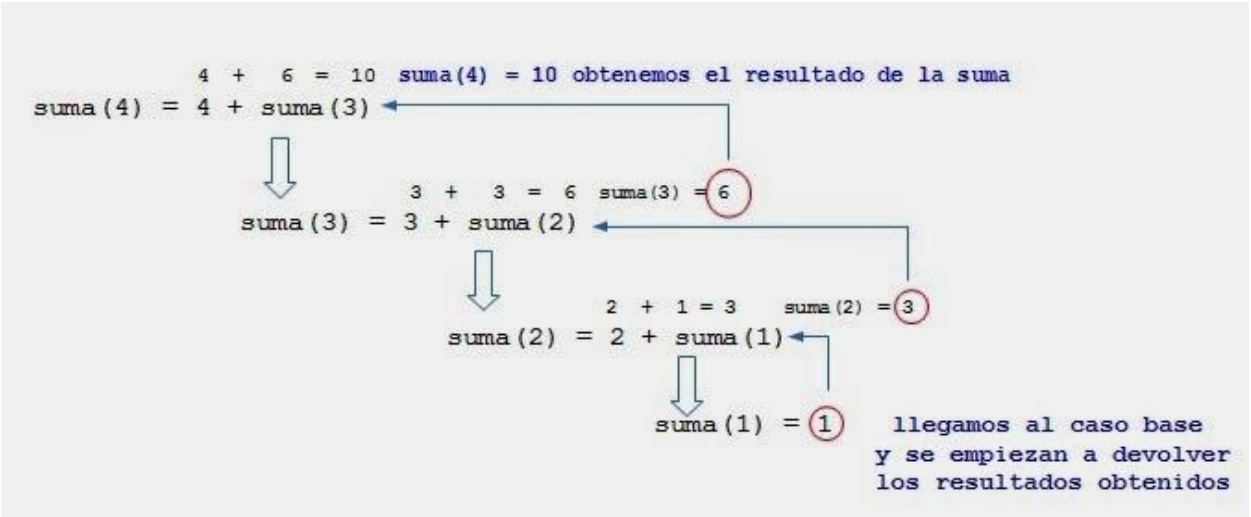
Se puede usar en toda situación en la cual la solución pueda ser expresada como una secuencia de movimientos, pasos o transformaciones gobernadas por un conjunto de reglas no ambiguas.

FUNCION RECURSIVA

Se componen de:

Caso base: una solución simple para un caso particular (puede haber más de un caso base).

Caso recursivo: una solución que involucra volver a utilizar la función original, con parámetros que se acercan más al caso base.



REGLAS

Un algoritmo recursivo debe tener un caso base.

En el contexto de un algoritmo recursivo, la recursión implica que la función se llama a sí misma para abordar subproblemas más pequeños. El caso base es esencial para evitar un bucle infinito y garantizar que el algoritmo llegue eventualmente a una solución o resultado final.

Un algoritmo recursivo debe cambiar su estado y moverse hacia el caso base.

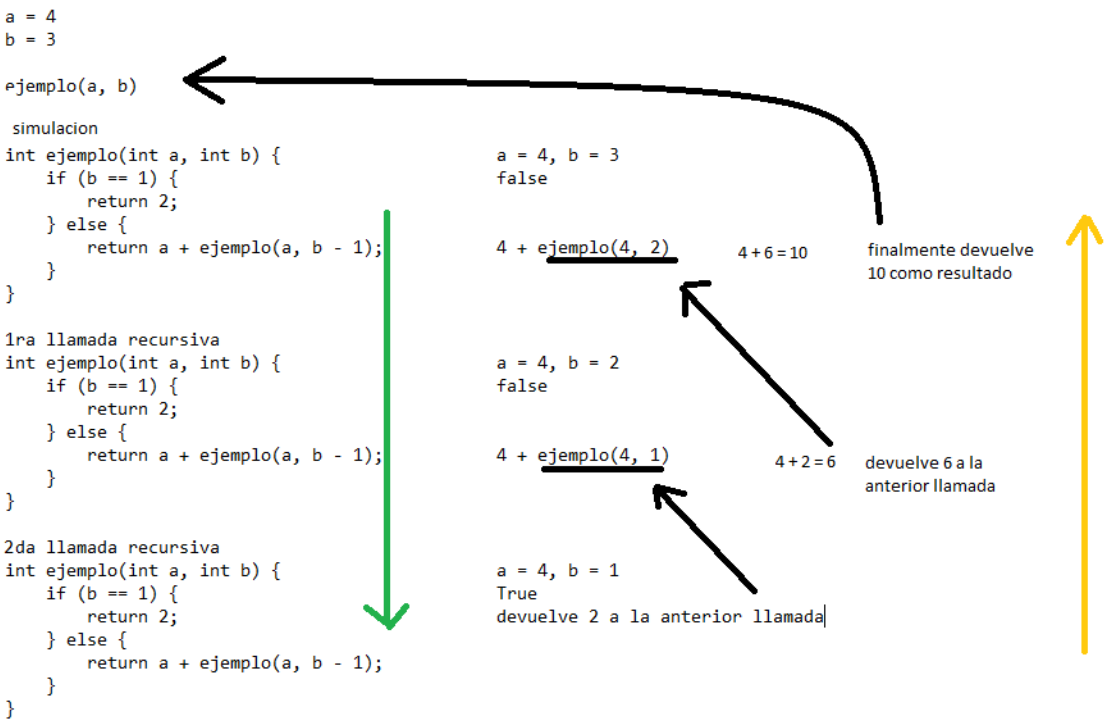
El algoritmo debe realizar operaciones que reduzcan el tamaño o la complejidad del problema en cada llamada recursiva, llevándolo gradualmente hacia el caso base. Este proceso de cambio de estado es crucial para asegurar que la recursión eventualmente alcance el caso base y evite un bucle infinito.

Un algoritmo recursivo debe llamarse a sí mismo, recursivamente.

La recursividad implica que la función se llama a sí misma, y esta llamada repetida conduce al abordaje de instancias más pequeñas del problema original. Cada llamada recursiva contribuye a la resolución del problema general, y este proceso continúa hasta que se alcanza un caso base que proporciona una solución directa sin la necesidad de más llamadas recursivas.

Enlace 1:

https://www.uv.mx/personal/ocastillo/files/2011/04/Recursividad.pdf

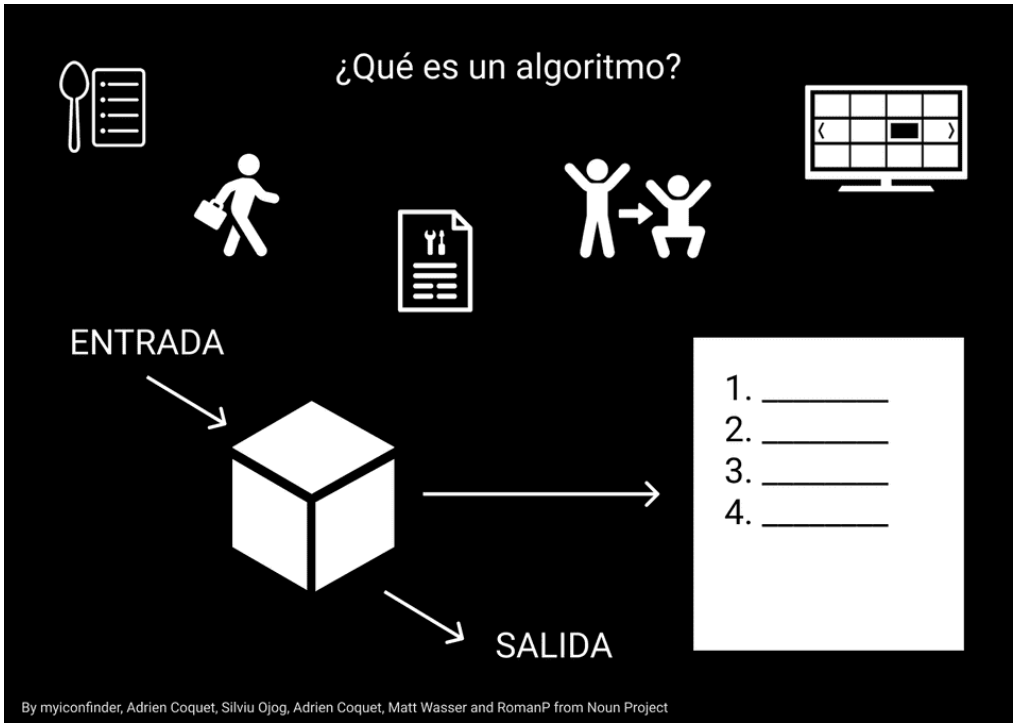


Enlace 2:
https://runestone.academy/ns/books/published/pythoned/Recursion/LasTresLeyesDeLaRecursividad.html

CONTENIDOS FUNDAMENTALES SOBRE EL ANÁLISIS, COMPLEJIDAD DE UN ALGORITMO Y LAS REGLAS BÁSICAS PARA DETERMINAR EL ANÁLISIS Y LA COMPLEJIDAD DE UN ALGORITMO

Un algoritmo es un conjunto de pasos o procedimientos utilizados para resolver un problema. Es un proceso bien definido que toma alguna entrada, realiza ciertas operaciones en la entrada y produce alguna salida. Los algoritmos son una parte importante de la informática y se utilizan para realizar una amplia variedad de tareas, desde cálculos simples hasta análisis de datos complejos e inteligencia artificial.
Enlace 1: https://www.europeanvalley.es/noticias/la-complejidad-de-los-algoritmos/

El análisis de algoritmos es una herramienta para hacer la evaluación del diseño de un algoritmo, permite establecer la calidad de un programa y compararlo con otros que puedan resolver el mismo problema, sin necesidad de desarrollarlos. El análisis de algoritmos estudia, desde un punto de vista teórico, los recursos computacionales que requiere la ejecución de un programa, es decir su eficiencia (tiempo de CPU, uso de memoria, ancho de banda, ...). Además de la eficiencia en el desarrollo de software existen otros factores igualmente relevantes: funcionalidad, corrección, robustez, usabilidad, modularidad, mantenibilidad, fiabilidad, simplicidad y aún el propio costo de programación.



El análisis de algoritmos se basa en:

El análisis de las características estructurales del algoritmo que respalda el programa.
La cantidad de memoria que utiliza para resolver un problema.
La evaluación del diseño de las estructuras de datos del programa, midiendo la eficiencia de los algoritmos para resolver el problema planteado.
Determinar la eficiencia de un algoritmo nos permite establecer lo que es factible en la implementación de una solución de lo que es imposible.

- Tipos de Análisis:
Peor caso (usualmente): T(n) = Tiempo máximo necesario para un problema de tamaño n
Caso medio (a veces): T(n) = Tiempo esperado para un problema cualquiera de tamaño n
· (Requiere establecer una distribución estadística)
Mejor caso (engañoso): T(n) = Tiempo menor para un problema cualquiera de tamaño n
Los criterios para evaluar programas son diversos: eficiencia, portabilidad, eficacia, robustez, etc. El análisis de complejidad está relacionado con la eficiencia del programa. La eficiencia mide el uso de los recursos del computador por un algoritmo. Por su parte, el análisis de complejidad mide el tiempo de cálculo para ejecutar las operaciones (complejidad en tiempo) y el espacio de memoria para contener y manipular el programa más los datos (complejidad en espacio). Así, el objetivo del análisis de complejidad es cuantificar las medidas físicas: tiempo de ejecución y espacio de memoria y comparar distintos algoritmos que resuelven un mismo problema.

El tiempo de ejecución de un programa depende de factores como:

- los datos de entrada del programa
- la calidad del código objeto generado por el compilador
- la naturaleza y rapidez de las instrucciones de máquina utilizadas
- la complejidad en tiempo del algoritmo base del programa

El tiempo de ejecución debe definirse como una función que depende de la entrada; en particular, de su tamaño. El tiempo requerido por un algoritmo expresado como una función del tamaño de la entrada del problema se denomina complejidad en tiempo del algoritmo y se denota T(n). El comportamiento límite de la complejidad a medida que crece el tamaño del problema se denomina complejidad en tiempo asintótica. De manera análoga se pueden establecer definiciones para la complejidad en espacio y la complejidad en espacio asintótica.

Entrada: *n*	
Salida: *c*	

Repetición(*n*){
1: *c* = 0;
2: **for** *i* = 1 **to** *n* **do**
3: *c*++;
4: **end for**
5: **return** *c*;
6: }
	int Repeticion(int n){
	1. int c=0;
	2. for(i=1; i<=n; i++)
	3. c++;
	4. return c;
	}

El tiempo de ejecución depende de diversos factores. Se tomará como más relevante el relacionado con la entrada de datos del programa, asociando a un problema un entero llamado tamaño del problema, el cual es una medida de la cantidad de datos de entrada.

La complejidad de un algoritmo de tamaño n es distinta dependiendo de las instancias de tamaño n del problema que resuelve. Esto nos lleva a estudiar la complejidad del peor caso, mejor caso, y caso promedio.

Para un tamaño dado (n), la complejidad del algoritmo en el peor caso resulta de tomar el máximo tiempo (complejidad máxima) en que se ejecuta el algoritmo, entre todas las instancias del problema (que resuelve el algoritmo) de tamaño n; la complejidad en el caso promedio es la esperanza matemática del tiempo de ejecución del algoritmo para entradas de tamaño n, y la complejidad mejor caso es el menor tiempo en que se ejecuta el algoritmo para entradas de tamaño n. Por defecto se toma la complejidad del peor caso como medida de complejidad T(n) del algoritmo.

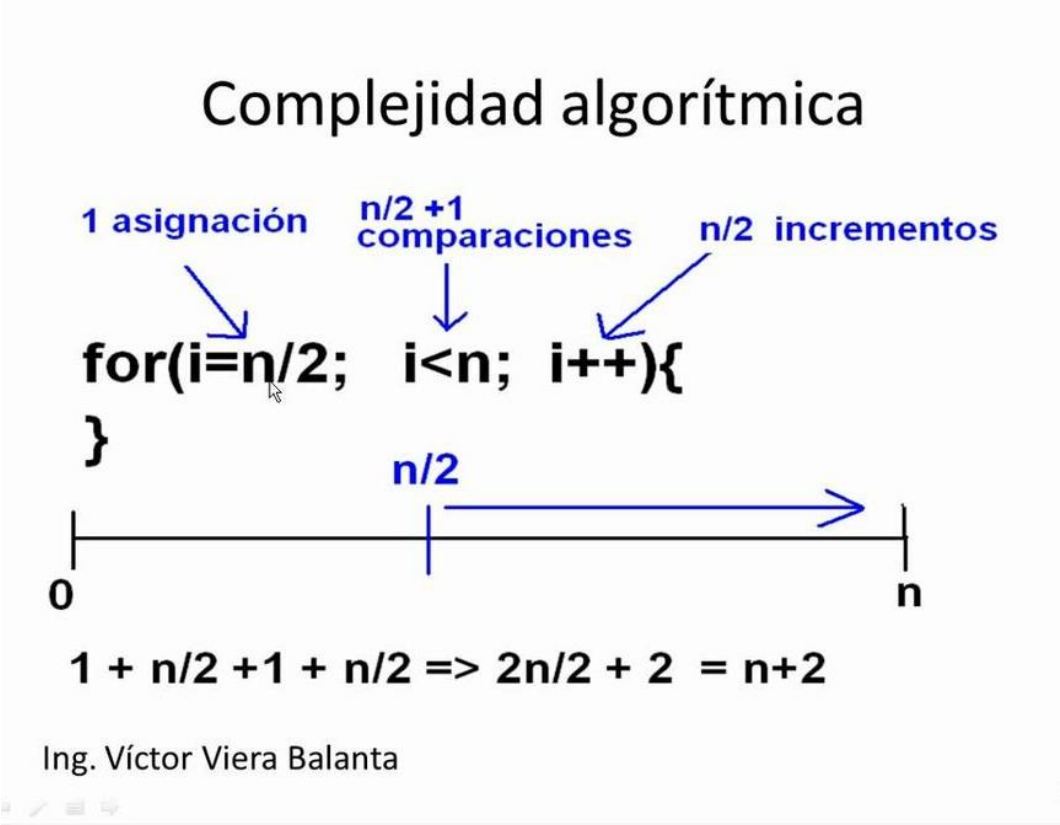
Aun cuando son los programas que los llegan realmente a consumir recursos computacionales (memoria, tiempo, etc.) sobre una máquina concreta, se realiza el análisis sobre el algoritmo de base, considerando que se ejecuta en una máquina hipotética.

La complejidad de un programa depende de:

La máquina y el compilador utilizados
El tamaño de los datos de entrada que depende del tipo de datos y del algoritmo
El valor de los datos de entrada.
Complejidad temporal: Se define como el tiempo que tarda un algoritmo en una entrada de tamaño n.

La complejidad en el caso peor proporciona una medida pesimista, pero fiable.

Dado que se realiza un estudio teórico de la complejidad, ignorando aspectos como las características de la máquina y el compilador, se tiene en cuenta que las diferencias en eficiencia se hacen más significativos para tamaños grandes de los datos de entrada se analiza la complejidad en términos de su comportamiento asintótico, dejando de lado la forma exacta de la función de complejidad.



Enlace 2: http://artemisa.unicauca.edu.co/~nediaz/EDDI/cap01.htm

REGLAS

1. Notación Big O: (La más importante)
Definición: La notación Big O describe el límite superior asintótico del tiempo de ejecución de un algoritmo en función del tamaño de la entrada.
2. Complejidad Temporal y Espacial:
Temporal: Se refiere al tiempo que tarda un algoritmo en ejecutarse en función del tamaño de la entrada.
Espacial: Se refiere a la cantidad de memoria que utiliza un algoritmo en función del tamaño de la entrada.
3. Tipos de Complejidad Temporal:
Constante (O(1)): El tiempo de ejecución no depende del tamaño de la entrada.
Lineal (O(n)): El tiempo de ejecución crece linealmente con el tamaño de la entrada.
Logarítmica (O(log n)): Común en algoritmos de búsqueda binaria.
Cuadrática (O(n^2)): Común en algoritmos de ordenamiento burbuja.
Linealítmica (O(n log n)): Común en algoritmos de ordenamiento eficientes como el Merge Sort y el Quick Sort.
4. Reglas Básicas para Determinar la Complejidad:
Iteración: El número de iteraciones en bucles.
Recursividad: El número de llamadas recursivas y su complejidad.
Operaciones Básicas: Contar las operaciones fundamentales dentro del algoritmo.

Evaluación en el Peor Caso: Analizar el rendimiento en el peor escenario posible.
5. Análisis Asintótico:
Enfoque: Se centra en el comportamiento del algoritmo a medida que el tamaño de la entrada tiende hacia infinito.
Importancia: Proporciona una visión general de la eficiencia del algoritmo sin preocuparse por detalles específicos.
6. Ejemplo Práctico:
Algoritmo de Búsqueda Binaria:
Complejidad Temporal: debido a su naturaleza de dividir a la mitad en cada iteración.
7. Uso de Estructuras de Datos Eficientes:
Elección adecuada: Utilizar estructuras de datos eficientes (listas enlazadas, árboles, tablas hash) puede influir en la complejidad.
8. Optimización vs. Legibilidad:
Balance: Es importante encontrar un equilibrio entre la optimización y la legibilidad del código.
9. Análisis Experimental:
Pruebas Prácticas: Complementar el análisis teórico con pruebas prácticas para validar la eficiencia en entornos reales.
10. Adaptabilidad a Escalas:
Eficiencia Escalable: Considerar cómo el algoritmo maneja tamaños de entrada cada vez mayores.

Enlace 3: https://www.campusmvp.es/recursos/post/Rendimiento-de-algoritmos-y-notacion-Big-O.aspx

ENLACE DE LAS DIAPOSITIVAS:
https://www.canva.com/design/DAF25Nlu4D0/reNJskadfJ1uNNUHJ-bTZA/edit?utm_content=DAF25Nlu4D0&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

ENTREGABLES:

- Una vez culminada tu tarea, súbela en este apartado del aula virtual “S4-Tarea-4:
- Este formato en pdf con el contenido de la investigación, con texto, imágenes, ejemplos con código.
- Subir la presentación en power point, canvas o prezi, etc.
- Recuerda el nombre del archivo deberá ser: **Tarea5_Algoritmos_2023B_NApellido**.

RECURSOS NECESARIOS

- Acceso a Internet.
- Imaginación.

- VSC