A – Introducción. Conceptos Básicos.

Definiciones

Informática: Es la ciencia que estudia el análisis y resolución de problemas utilizando computadoras.

Ciencia: Metodología fundamentada y racional para el estudio y resolución de los problemas.

Computadora: Es una máquina digital y sincrónica con cierta capacidad de cálculo numérico y lógico, controlada por un programa almacenado y con posibilidad de comunicación con el mundo exterior. Digital porque representa la información con valores binarios. Sincrónica porque realiza las operaciones controlada por un reloj. Internamente tiene instrucciones almacenadas que podrá leer interpretar y ejecutar.

Programa: Es un conjunto de instrucciones ejecutables sobre una computadora que permite cumplir una función específica. Un programa sin errores puede ser incorrecto si no cumple con lo solicitado en los requerimientos. Es correcto si cumple lo solicitado en los requerimientos.

Dato: Es la representación de un objeto del mundo real mediante la cual se pueden modelar aspectos de un problema que se desea resolver con un programa sobre una computadora. Representar datos requiere una transformación del mundo real a alguna forma de representación binaria que pueda ser interpretada por una computadora.

Tipo de dato: Es una clase de objetos de datos ligados a un conjunto de operaciones para crearlos y manipularlos.

Abstracción: Es el proceso de análisis del mundo real para interpretar los aspectos esenciales de un problema y expresarlo en términos precisos.

Modelización: Es el proceso de abstraer un problema del mundo real y simplificar su expresión, tratando de encontrar los aspectos principales que se pueden resolver (requerimientos), los datos que se deban procesar y el contexto del problema.

Precondición: Es toda información que se conoce como verdadera antes de iniciar el programa.

Postcondición: Es toda información que debería ser verdadera al concluir un programa, si se cumple adecuadamente el requerimiento pedido.

Especificación: Es el proceso de analizar los problemas del mundo real y determinar en forma clara y concreta el objetivo que se desea. Establecer en forma unívoca el contexto, las precondiciones y el resultado esperado, del cual se derivan las postcondiciones.

Lenguaje de programación: Es el conjunto de instrucciones permitidas y definidas por sus reglas sintácticas y su valor semántico, para la expresión de soluciones a problemas.

Algoritmo: Es la especificación rigurosa (clara y unívoca) de la secuencia de pasos a realizar sobre un autómata para alcanzar un resultado deseado en un tiempo finito (empieza y termina).

Hardware: Son los aspectos vinculados con la arquitectura física de las computadoras, las comunicaciones entre ellas y los dispositivos periféricos que utilizan.

Software: Es todo lo que se refiere a la programación básica y de aplicaciones de modo de dar una utilidad definida al hardware.

Eficiencia: Es una métrica de calidad de los algoritmos, asociada con una utilización óptima de los recursos del sistema de cómputo donde se ejecutará el algoritmo.

Modelización de problemas del mundo real.

El primer paso para la resolución de problemas por computadora involucra un ***análisis,*** con el objetivo de lograr establecer un modelo o abstracción del mundo real que sea el punto de partida para una solución por computadora.

El ***modelo*** debe permitirnos sintetizar los requerimientos del problema y simplificar el contexto y los datos a utilizar por el programa en la computadora.

Del problema real a su solución por computadora.

A partir de un problema del mundo real se realiza un ***análisis*** del problema realizando un proceso de ***abstracción*** para llegar al modelo, teniendo en cuenta el contexto del problema. Se deben obtener los requerimientos del usuario.

A partir del modelo se pasa a una etapa de ***diseño*** haciendo un análisis de la solución como sistema. De este análisis se realiza un proceso de ***modularización,*** una descomposición en módulos que tengan una función definida. Y luego de esta modularización de la solución o análisis se diseñan los algoritmos que luego darán origen al programa.

La descomposición funcional de todas las acciones que propone el modelo nos ayudará a reducir la complejidad, a distribuir el trabajo y en el futuro a re-utilizar los módulos.

En esta etapa se deben especificar el ***algoritmo*** para la solución del problema el cual puede no ser único y su elección es muy importante para la eficiencia posterior del sistema.

Una vez que se tiene el diseño se debe pasar a la etapa de ***implementación*** la cual requiere escribir los ***algoritmos*** en un lenguaje de programación específico y elegir la representación de los ***datos.***

Por último, una vez que se tiene un programa escrito en un lenguaje real y depurado de errores, debemos verificar que su ejecución conduce al resultado deseado, con datos representativos del problema real.

El análisis y diseño no dependen del lenguaje de programación. Se trata de entender el problema, modelizarlo, esquematizar su solución, modularizar y escribir los algoritmos.
La implementación depende del lenguaje de programación. Se trata de codificar los algoritmos, compilarlos a lenguaje de máquina, prueba en ejecución y verificación de cumplimiento de los requisitos.

Programas

Los componentes básicos de un programa son sus instrucciones y datos. Las instrucciones representan las operaciones que ejecutará la computadora al interpretar el programa. Los datos son los valores de información de los que se necesita disponer y en ocasiones transformar para ejecutar la función del programa.
Un tercer componente de los programas son los comentarios. Son aclaraciones sobre el código que no se interpretan.

Los datos los podemos clasificar como constantes (que no cambian) o como variables (que cambian a lo largo del programa). En relación a la modularización los podemos clasificar en globales (disponibles en todo el programa) o locales (disponibles solo en el módulo que se los define).

La memoria principal se divide en memoria de instrucciones y memoria de datos almacenándose en ellas las instrucciones y los datos respectivamente.

Los programas deben tener las siguientes características:

Para la audiencia humana:

- ***Operatividad:*** El programa debe realizar la función para la que fue concebido.

- ***Legibilidad:*** El código fuente de un programa debe ser fácil de leer y entender. Esto obliga a acompañar a las instrucciones con comentarios adecuados.

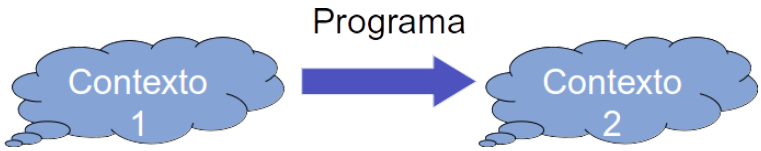
- ***Organización:*** El código de un programa debe estar descompuesto en módulos que cumplan las sub-funciones del sistema.

- ***Documentados:*** Todo el proceso de análisis y diseño del problema y su solución debe estar documentado mediante texto y/o gráficos para favorecer la comprensión, la modificación y la adaptación a nuevas funciones.

Para la CPU de la computadora:

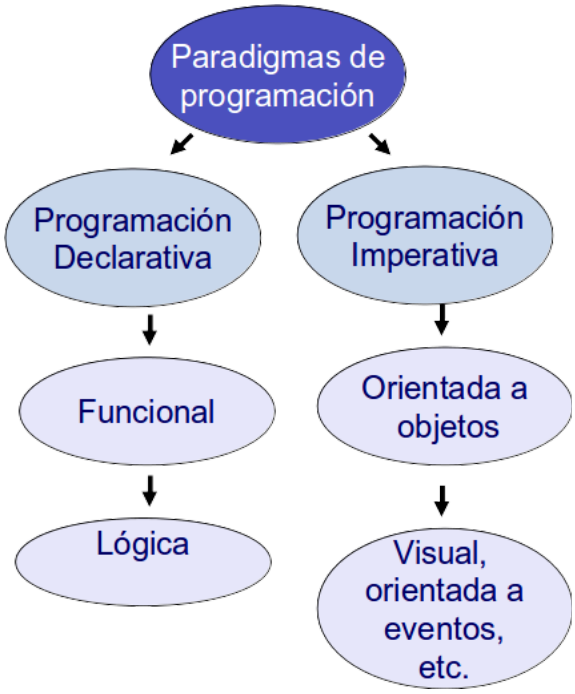
- Debe contener instrucciones válidas.
- Deben terminar.
- No deben utilizar recursos inexistentes.

A partir de un contexto determinado por las precondiciones, el programa transforma la información y debiera llegar al resultado esperado produciendo un nuevo contexto, caracterizado por las postcondiciones.



Paradigmas de programación

Los lenguajes de programación pueden ser clasificados a partir del modelo que siguen para ***definir*** y ***operar*** información. Este aspecto permite jerarquizarlos según el ***paradigma*** que siguen.



Programar

- Elegir la representación adecuada de los datos del problema.
- Elegir el lenguaje de programación a utilizar, según el problema y la máquina a emplear.
- Definir el conjunto de instrucciones (en el lenguaje elegido) cuya ejecución ordenada conduce a la solución.

B – Algoritmos. Acciones Elementales.

Estructuras de control. Modelo de máquina abstracta.

Secuencia: Representa una sucesión de operaciones que se ejecutan en el orden físico en que aparecen.
(figura)

Repetición: Representa la repetición de un bloque de acciones, un número N de veces fijo y conocido previamente. Continuará mientras no se cumplan las N veces.
La estructura de control que la representa es el **for** y debe contar con un índice ordinal autoincremental que cuente las repeticiones, estableciendo un valor inicial y un valor final. Así la cantidad de repeticiones es igual a *ValorFinal – ValorInicial + 1*
(figura)

Decisión: Representa la decisión entre dos alternativas en función de los datos del problema.
La estructura de control que la representa es el **if** , que evalúa una condición que retorna un valor lógico para tomar la decisión.
(figura)

Selección: Representa una decisión entre múltiples alternativas (más de dos).
La estructura de control que la representa es el **select**, **case**, **switch**. Se evalúa la variable decisión con el valor en cada posibilidad, retornando un valor lógico. En caso verdadero se ejecuta el bloque de acciones.
(figura)

Iteración: Representa la repetición de un bloque de código en función de una condición, repitiéndose una cantidad de veces desconocida. Esta estructura de control iterativa condicional puede ser *pre-condicional*, cuando se evalúa la condición antes de ejecutar el bloque de acciónes y esta representada por el **while**, o *post-condicional*, cuando se evalúa la condición luego de ejecutar el bloque de acciónes, el cual se ejecuta al menos una vez, y esta representada por el **do until**.
(figuras)

Corrección de algoritmos. Importancia de la verificación.

Antes de escribir un programa se debe tener una especificación completa y precisa de los requerimientos. Un programa es correcto cuando cumple con los requerimientos propuestos. Para asegurarse de esto es importante verificarlo con datos reales.

Eficiencia de un algoritmo.

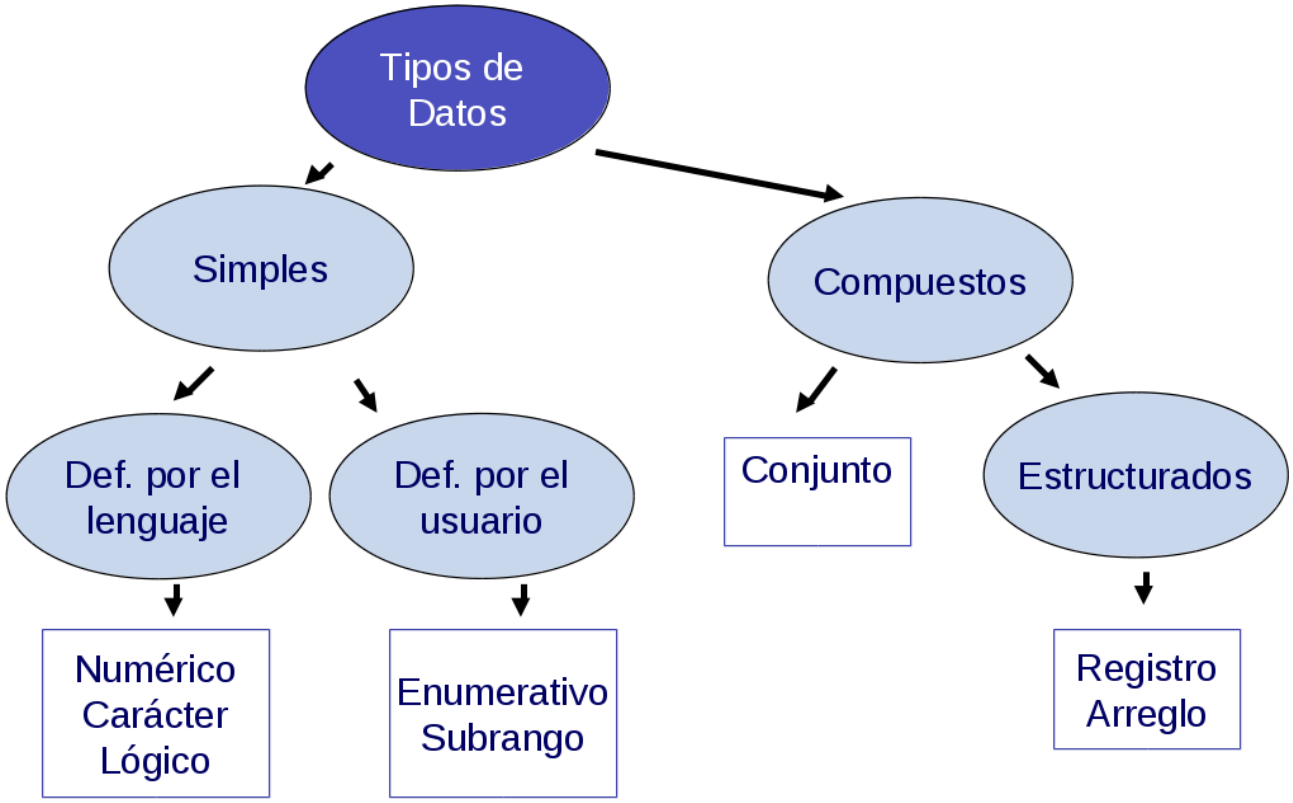
Importancia de la documentación de un algoritmo.

Es importante documentar un algoritmo para facilitar su lectura y mantenimiento. Comúnmente se hace por medio de comentarios en el mismo código.

C - Datos y Tipos de datos

Los ***datos*** son la representación de un objeto del mundo real mediante la cual se pueden modelar aspectos de un problema que se desea resolver con un programa sobre una computadora. Representar datos requiere una transformación del mundo real a alguna forma de representación binaria que pueda ser interpretada por una computadora.

Los ***tipos de datos*** son una clase de objetos ligados a un conjunto de operaciones para crearlos y manipularlos. Se caracterizan por un rango de valores posibles, un conjunto de operaciones realizables sobre ese tipo y su representación interna. Al definir un tipo de dato se está indicando la clase de valores que pueden tomar sus elementos y las operaciones que se pueden realizar sobre ellos.



Constantes y variables.

Un dato de tipo ***constante*** denota un dato que no cambia que puede ser numérico, lógico o carácter.

Un dato de tipo ***variable*** denota un dato que puede cambiar que puede ser numérico, lógico o carácter.

Los datos, ya sean constantes o variables se definen asociandolos a identificadores que son nombres descriptivos que nos permiten abstraernos de la dirección de memoria del objeto y su valor.
Algunos lenguajes denominados "fuertemente tipados", exigen que se especifique a que tipo pertenecen cada una de las variables, y verifican que el tipo de los datos asignados a esa variable se corresponda con su definición.
Otros lenguajes denominados "auto tipados" verifican el tipo de las variables según su nombre.

Y otros denominados “dinámicamente tipados” permiten que una variable tome valores de distinto tipo.
Una de las sentencias básicas que poseen los lenguajes de programación es la asignación, que consiste en evaluar la expresión ubicada a la derecha del operador y almacenarla en la variable ubicada a la izquierda.

Tipos de datos simples y compuestos.

Los tipos de datos ***simples*** son aquellos que toman un único valor, en un momento determinado, de todos los permitidos para ese tipo.

Los tipos de datos ***compuestos*** pueden tomar varios valores a la vez que guardan alguna relación lógica entre ellos.

Tipos de datos primitivos (definidos por el lenguaje).

Los tipos de datos ***definidos por el lenguaje*** (primitivos o estándar) son provistos por el lenguaje y tanto la representación como sus operaciones y valores son reservadas al mismo.

NUMÉRICO

El tipo de dato ***numérico*** es el conjunto de los valores numéricos, que pueden representarse en enteros o reales.

El tipo de dato numérico ***entero*** es el más simple. De acuerdo a la cantidad de bits que se utilicen para representar un entero, vamos a tener una cantidad finita de valores que se pueden representar, y así un máximo y un mínimo.

El tipo de dato numérico ***real*** permite representar números decimales, que forman una serie ordenada desde un valor mínimo negativo hasta un valor máximo determinado, pero que no están distribuidos de manera uniforme.
Este tipo de dato tiene una representación finita con una precisión fija de números significativos.
Esta representación se denomina ***coma flotante*** que consiste en definir cada número como una mantisa y un exponente.

Las operaciones válidas entre operandos, para el tipo de dato numérico y sus operadores aritméticos son suma(+), resta(-), multiplicación(*), división(/), división entera(div), módulo(mod).

Cuando se están realizando operaciones puede ocurrir que el resultado supere el máximo valor permitido por la representación, lo que se denomina ***overflow.***

Puede ocurrir que para resultados con valores demasiado pequeños este por debajo del límite permitido, lo que se denomina ***underflow.***

Para expresiones complejas con más de dos operandos las operaciones se ejecutan según un orden de precedencia, el cual se puede alterar usando paréntesis. 1º *, /. 2º +, -. 3º mod, div.

El tipo de dato numérico también acepta operadores relacionales que permiten comparar valores dando por resultado un tipo de dato lógico.
Igualdad(=), desigualdad(<>), de orden(<, <=, >, >=).

LÓGICO

El tipo de dato ***lógico,*** también llamado booleano puede tomar un valor entre un conjunto de dos posibilidades true (verdadero) o false (falso), utilizándose en casos donde se representan dos alternativas a una condición, y dando también como resultado un valor lógico. Negación(not), conjunción(and), disyunción(or). El orden de precedencia es not, and, or.

CARÁCTER

El tipo de dato ***carácter*** proporciona objetos de un conjunto de elementos establecido y normalizado por el estándar ASCII, que establece los elementos y su orden de precedencia, conteniendo sólo un elemento como su valor que puede ser letra, número o símbolo.

El tipo de dato carácter también acepta operaciones relacionales para comparar valores (=, <>, <=, <, >=, >) dando como resultado un valor booleano. Esto es posible porque el código ASCII define un valor para cada elemento.

Tipos de datos definidos por el usuario.

La representación de los fenómenos reales que se tratan de resolver utilizando computadoras requiere elementos que no son tipos estándar de un lenguaje. Entonces se brinda la posibilidad de representar datos no estándares, especificando sus valores permitidos y las operaciones válidas sobre los mismos.

Los tipos de datos ***definidos por el usuario***, permiten definir nuevos tipos de datos a partir de los tipos simples. No existe en la definición del lenguaje y el usuario es el encargado de determinar su denominación y el conjunto de valores y operaciones que dispondrá el mismo.
Los lenguajes de programación permiten asignar nombres a los tipos y usarlos en declaraciones posteriores. Son identificadores que pueden ser utilizados en declaraciones de tipos variables, procedimientos y funciones.

ENUMERATIVO

El tipo de dato ***enumerativo*** es un grupo de valores nombrados definidos por el usuario. Se suelen definir encerrando los valores entre paréntesis y separándolos por comas.
Los identificadores que detallan los valores posibles del tipo, son constantes simbólicas que no se permiten en más de una definición de tipo enumerativo.
Las operaciones que se permiten sobre este tipo son de asignación y comparación.
Las funciones predefinidas son aquellas predefinidas para los tipos ordinales.

SUBRANGO

El tipo de dato ***subrango*** es un tipo ordinal que consiste en una secuencia contigua de valores de algún tipo ordinal. Se utilizan cuando no son necesarios todos los valores de un tipo ordinal, sino un subrango de ellos.
Las operaciones y funciones predefinidas para el tipo subrango son las mismas que para el tipo base.

CONJUNTO

El tipo de dato ***conjunto*** es una colección de datos simples del mismo tipo sin repetición ni orden entre ellos.
Un conjunto se constituye definiendo los elementos individuales consecutivamente separados por comas y todos encerrados entre corchetes pudiendo ser asignados a variables de tipo conjunto y usarse como operandos en ciertos tipos de expresiones lógicas.
Un conjunto sin elementos es un conjunto vacío.
Las operaciones sobre datos de tipo conjunto son la unión (+), intersección (*), diferencia (-), pertenencia (in).

STRING

El tipo de dato ***string*** es una sucesión de caracteres que se almacena en un área contigua de memoria y que puede ser leído o escrito.
Se debe indicar el tamaño máximo que se desea manejar. Si no se especifica podrá contener como máximo 255 caracteres y se asigna igual que un carácter.
Las operaciones permitidas sobre el tipo string son asignación, concatenación y operaciones lógicas.
Las funciones predefinidas son copy(string,posición, cantidad), length(string), pos(string1,string2), concat(string1,string2,...).

Funciones predefinidas.

Son funciones predefinidas por el lenguaje para aplicar a los tipos de datos.

Tipos ordinales.

Los tipos de datos ***ordinales*** son un tipo de dato ordenado discretamente donde para cada elemento que es parte del tipo existe un elemento anterior y otro posterior.
Los tipos de datos entero, carácter y lógico son tipos de datos ordinales.
El tipo de dato real no es ordinal ya que entre dos números reales siempre habrá otro número real.

D - Modularización. Procedimientos y funciones. Parámetros.

Descomposición de problemas. Utilidad e importancia.

Modularizar significa dividir un problema en partes funcionalmente independientes, que encapsulen operaciones y datos, y que desarrollen una tarea específica. Cada una de estas partes se denomina Módulo y representan tareas específicas bien definidas que deben comunicarse entre sí adecuadamente y cooperar para conseguir un objetivo común.

La descomposición de un problema en subproblemas debe ser de forma tal que, cada subproblema esta a un mismo nivel de detalle, cada subproblema puede resolverse independientemente del resto y las soluciones de estos subproblemas pueden combinarse para resolver el problema original.

La noción de reusabilidad.

Subprogramas o módulos.

Un ***subprograma*** o ***módulo*** es la abstracción y encapsulación de una tarea del programa que puede desarrollarse de manera independiente.

Un módulo puede realizar las mismas tareas que un programa pero solo se ejecuta cada vez que el programa principal u otro subprograma lo invoca.
Cada módulo debe tener un nombre que lo identifique y con el cual se lo invoca.
Un módulo conoce todo lo que esta definido anteriormente a él y todo lo que está definido inmediatamente en su interior.
Así un módulo puede invocar a otro módulo que esté definido en su contexto que depende de su ubicación dentro de la definición de los módulos.

Procedimientos.

Un ***procedimiento*** es un conjunto de instrucciones que realizan una tarea específica y como resultado puede retornar o no uno o más valores como respuesta.
Cuando se invoca al procedimiento se escribe como una instrucción el nombre del mismo y entre paréntesis se indica la lista de parámetros.

Funciones.

Una ***función*** es un módulo que a partir de uno o más valores realiza una tarea específica y retorna un único valor como resultado.
La función indica el tipo de dato que retorna como resultado. El resultado debe ser asignado a una variable, impreso en pantalla o debe ser usado como parte de una condición lógica.

Conceptos de argumentos y parámetros.

Los ***parámetros*** son la serie de datos con la que se comunican los módulos. Cada parámetro debe especificar el tipo de dato con el que se corresponde. Estos parámetros definidos en el encabezado son ***parámetros formales*** y los parámetros que se definen en el llamado del módulo reciben el nombre de ***parametros actuales***. Ambos deben coincidir en tipos y cantidad.

Existe una correspondencia entre los parámetros formales y actuales que se puede resolver por ***posición*** y por ***nombre.***
Cuando se corresponden por posición significa que la posición de un dato en los parámetros actuales equivale a la posición en los parámetros formales.
Cuando se corresponden por nombre en la invocación del módulo se permiten indicar con qué parámetro formal se corresponde cada parámetro actual.
Los métodos para el pasaje de parámetros son formas mediante las cuales los parámetros actuales y formales son transmitidos o devueltos entre los módulos.

El pasaje de parámetros por ***valor*** consiste en inicializar el parámetro formal correspondiente con el valor del parámetro actual, que luego actúa como una constante local del subprograma. Este parámetro se denomina de entrada.

En el pasaje de parámetros por ***resultado*** no se transmite ningún valor en el llamado del módulo, sino que al finalizar se copia el valor del parámetro formal en el parámetro actual correspondiente, transmitiendo un resultado al módulo que llama. Este parámetro se denomina de salida.

En el pasaje de parámetros por ***valor-resultado,*** el valor del parámetro actual es utilizado para inicializar el parámetro formal correspondiente, el cual puede utilizarse como variable local del módulo y posteriormente su valor es retornado al parámetro actual. Este parámetro se denomina de entrada-salida.

En el pasaje de parámetros por ***referencia*** en vez de transmitirse un dato se transmite la dirección real de memoria donde la variable se encuentra. Así el parámetro actual y formal comparten la misma zona de memoria, por esto cualquier modificación se realiza sobre la misma variable.

Conceptos de variables locales y variables globales.

Una de las características que poseen las variables es el alcance o ámbito de referencia de las mismas, que determina la zona del programa donde la variable es definida y conocida. Así entonces tenemos variables ***locales*** y ***globales.***

Una variable ***local*** es aquella que está declarada y definida dentro de un programa o módulo, en el sentido que esta dentro de ese módulo y es distinta a cualquier variable que tenga el mismo nombre y que estuviera declarada en otro lugar del programa.

Una variable ***global*** es aquella que está declarada en el programa y que puede ser utilizada en cualquier módulo de éste.

El uso de variables locales presenta la ventaja importante de construir módulos altamente independientes, donde las comunicaciones necesarias desde los programas que los invoquen deben hacerse mediante el pasaje de parámetros .

Es una buena práctica que favorece la reutilización y el mantenimiento del software.

Manejo de memoria en ejecución.

E - Estructuras de datos

Introducción y clasificación de las estructuras de datos.

Una ***estructura de datos*** es un conjunto de variables relacionadas entre si de diversas formas.

Todos los tipos estructurados son moldes para variables estructuradas que pueden tener más de un valor y pueden ser construidos a partir de tipos simples o de otros tipos estructurados.
Permiten trabajar con estructuras de datos compuestas para representar los elementos del mundo real.

Una estructura de datos ***simple*** es una estructura que representa un único valor.

Una estructura de datos ***compuesta*** es una estructura que puede contener más de un valor.

Una estructura de datos ***homogénea*** es una estructura donde los datos que la componen son todos del mismo tipo.

Una estructura de datos ***heterogénea*** es una estructura donde los datos que la componen son de distinto tipo.

Una estructura de datos ***estática*** es una estructura donde la cantidad de elementos que contiene es fija. La cantidad de memoria que utiliza no varía durante la ejecución del programa.

Una estructura de datos ***dinámica*** en una estructura donde el número de componentes, y por lo tanto la cantidad de memoria, puede variar durante la ejecución de un programa.

Registros.

Los registros permiten agrupar datos de diferentes clases y con una conexión lógica en una estructura única.

Un ***registro*** es un conjunto de valores que puede ser de distinto tipo, donde cada valor tiene un identificador y se lo denomina *campo*. El almacenamiento ocupado es fijo.
Así un registro es una estructura heterogénea y estática.

Se construye definiendo al tipo como un registro y especificando el nombre y tipo de los campos.

```
type
     registro = record
                     campo1: tipo1;
                     campo2: tipo2;
                     …
                end;
```

Cuando se desea acceder a un campo de un registro hay que especificar el nombre del registro y el nombre del campo. Esto se denomina ***calificar*** el campo.

Las variables estructuradas como los registros se pueden anidar. Un campo de un registro puede ser a su vez otro registro.

Las operaciones posibles para los campos de un registro son las mismas que para el tipo correspondiente a esos campos.
Y sobre todo el registro se puede operar la asignación, si las variables usadas son del mismo tipo de registro.

Dos registros no pueden ser comparados. Se debe hacer campo por campo.
Tampoco se puede leer o escribir un registro completo, también se debe hacer campo por campo.

Cuando se trabaja con registros se puede simplificar el acceso a los campos con la sentencia `with` que permite que un registro sea nombrado una vez y luego sea accedido directamente.

Arreglos.

La estructura de datos ***arreglo*** es una colección (estructura compuesta) ordenada e indexada de elementos donde todos los elementos son del mismo tipo (estructura homogénea) y pueden recuperarse en cualquier orden indicando la posición que ocupan en la estructura (estructura indexada) a través de una variable índice.
Esta estructura se compone de un conjunto de variables que se almacenan consecutivamente en memoria y su espacio ocupado durante la ejecución del programa es fijo (estructura estática).

Vectores.

Un ***vector*** o ***arreglo lineal*** es un tipo de dato con un índice, es decir, una sola dimensión.

Para acceder a los elementos

```
vector[indice]
```

Para definir un vector

```
type nombre = array[indice] of tipo_de_dato;
```

El nombre es el identificador asociado a un área de memoria fija y consecutiva. El índice es el rango de valores que puede tomar el índice. El tipo de dato es el tipo de dato de los elementos del vector.

Para asignar un valor

```
vector[indice]:= valor
```

Operaciones con vectores.

Recorrer un vector

Buscar un elemento

Acceder a una posición

Agregar un elemento

En la operación de agregado de un elemento se debe verificar en primer lugar que el vector tenga espacio disponible para almacenar el nuevo elemento, verificando que su dimensión lógica sea menor que su dimensión física. Si se dispone de lugar en el vector, se agrega el elemento al final del vector, en la posición de la dimensión lógica más uno.

Insertar un elemento

En la operación de inserción de un elemento se debe verificar en primer lugar que el vector tenga espacio disponible para almacenar el nuevo elemento verificando que su dimensión lógica sea menor que su dimensión física.
Si se dispone de lugar en el vector se debe tener en cuenta el caso que se deba insertar en una posición dada o por algún criterio de orden.

Al insertar un elemento en una posición dada se debe verificar que esa posición sea una posición válida, o sea que se encuentre entre la primer posición y la dimensión lógica más uno.

Al insertar un elemento por algún criterio de orden se debe recorrer el vector desde la primer posición hasta encontrar la posición donde se cumpla con el criterio o llegar al final del vector.

En ambos casos si el elemento se debe agregar al final se inserta directamente, en cambio si el elemento se debe agregar en una posición intermedia donde la posición se encuentra ocupada, se debe abrir espacio en el vector, para el nuevo elemento, generando un corrimiento de los elementos ubicados entre la posición a insertar y el último elemento hacia su posición inmediata siguiente iniciando desde el último elemento.

Una vez que se liberó el espacio se inserta el elemento en la posición indicada y se incrementa en uno el valor de la dimensión lógica.

Borrar un elemento

En la operación de borrado de un elemento se debe tener en cuenta el caso que se deba borrar un elemento en una posición específica o borrar un elemento dado.

En el caso que se disponga de la posición se debe verificar que la posición sea válida, o sea que se encuentre entre la primer posición y la dimensión lógica.

En el caso que se disponga del elemento se debe recorrer el vector hasta encontrar el elemento y así obtener su posición o hasta llegar al final del vector comprobando que no existe el elemento.

Una vez hallada la posición a borrar se debe hacer un corrimiento de todos los elementos ubicados entre la posición siguiente al elemento a borrar y el último elemento (dimensión lógica) del vector, desde la posición en la que se encuentran hacia su posición anterior y comenzando desde el elemento siguiente al elemento a borrar.

Por último, una vez eliminada la posición se debe decrementar en uno la dimensión lógica.

Matrices. Tratamiento de información estructurada en vectores y matrices.

Una ***matriz*** es un tipo de dato arreglo con dos dimensiones o índices. Es un grupo de elementos homogéneo, con un orden interno y en el que se necesitan dos índices para referenciar un único elemento de la estructura.

```
matriz[fila,columna]
```

Para definir una matriz

```
type nombre = array[índice1,índice2] of tipo_de_dato;
```

El índice1 es el rango de valores para las filas y el índice2 es el rango de valores para las columnas.

Para asignar un valor en una matriz

```
matriz[pos1,pos2] := valor;
```

Siguiendo la lógica de vectores y matrices se puede representar arreglos de N dimensiones, restringido por el lenguaje y el espacio que ocupe en memoria.

Un arreglo puede ser enviado como parámetro a un módulo o recibirse como respuesta. No siempre es aconsejable enviarla por valor ya que el dato se copia y si es una estructura muy grande no es eficiente en el uso de recursos.

Algoritmos de búsqueda.

Dada una colección de elementos se pueden obtener dos tipos de información.

- Información relativa al conjunto de datos de dicha colección.
- Información detallada de algún elemento en particular.

Para este último caso es necesario ubicar y extraer el elemento.

Un ***algoritmo de búsqueda*** es el proceso de ubicar información particular en una colección de datos.

La forma de buscar información depende de cómo están organizados los datos. Así es que reordenando la información es posible mejorar la eficiencia de búsqueda.

Búsqueda lineal

Cuando no se tiene información sobre como estan organizados los elementos se puede buscar desde el inicio analizando elemento por elemento hasta encontrarlo o llegar al final.
También se puede incorporar el elemento al final y si no se encuentra extender la longitud (centinela).

Búsqueda binaria

Cuando los elementos están ordenados de forma creciente se podría hacer una búsqueda lineal hasta encontrar el elemento o uno mayor. Pero también se podría hacer una búsqueda binaria.
En esta se compara el elemento buscado con el elemento ubicado a la mitad de la colección. Si son iguales, la búsqueda termina, si el elemento es menor se continúa la búsqueda en la primer mitad y en caso contrario se continúa la búsqueda en la segunda mitad. Así se va ignorando el 50% restante que no es necesario analizar, mejorando la eficiencia.

Algoritmos de ordenación. Ordenación por índice.

Un ***algoritmo de ordenación*** es el proceso por el cual se puede ordenar una colección de elementos.

Método de selección

Este método selecciona el menor elemento y lo intercambia con el primero, luego busca entre los restantes el menor y lo intercambia con el elemento en la posición siguiente al último menor ordenado, y así hasta que todos estén ordenados. Se realizan n - 1 pasadas por el vector.

Método de intercambio o burbujeo

Este método no mueve los elementos grandes distancias, compara items adyacentes y los intercambia si están desordenados. Así al final de la primer pasada el elemento más grande queda al final. Se realizan n - 1 pasadas.
Se puede registrar si hubo cambios o no para ahorrar pasadas innecesarias cuando no hay cambios y está ordenado.

Método de inserción

El método de inserción ordena un vector insertando cada elemento a[i] entre los i - 1 anteriores, empezando por el segundo elemento.

F - Estructura de datos enlazadas: listas.

Alocación dinámica. Punteros.

Una ***lista*** es un conjunto de elementos del mismo tipo por lo cual es una estructura de datos *homogénea*, donde los mismos no ocupan posiciones secuenciales o contiguas de memoria, es decir, los elementos o componentes de una lista pueden aparecer físicamente dispersos en la memoria, aunque mantienen un orden lógico interno, ya que se generan dinámicamente en tiempo de ejecución, por lo cual es una estructura *dinámica.* Además es una estructura de datos *secuencial* donde cada elemento tiene un antecesor y un sucesor, y donde para acceder a una posición dada se debe recorrer la lista hasta la posición indicada.

Como los elementos que forman la lista, no están en posiciones contiguas, necesitan de un indicador que apunte al próximo elemento.

Un ***puntero*** es un tipo de variable, en la cual se almacena la dirección en memoria de un dato y permite manejar direcciones apuntando a un elemento determinado.

Así cada elemento de la lista va a estar constituido por el dato que se maneja y un puntero conteniendo la dirección del próximo elemento en la lista.

Para crear una lista se deben crear cada uno de sus nodos que luegos serán enlazados. Para esto se debe solicitar espacio en memoria para almacenar el

nodo. Se realiza con la instrucción new quedando almacenada la dirección de memoria en la variable puntero.

```
new (variable_puntero)
```

Así cada nodo de la lista será una variable de tipo registro que contendrá los datos y un puntero al nodo siguiente.

```
type
     pun = ^reg;
     reg = record
               nombre: string[50];
               siguiente: pun;
          end;

var
     pun1: pun;
     reg1: reg;
```

Para operar sobre el campo nombre del registro apuntado por `pun1`

```
pun1^.nombre
```

Para poder acceder a una lista se debe guardar en una variable la dirección al primer elemento. Para detectar el final de la lista se utiliza un puntero nulo (nil) que indica que no hay más elementos en la lista.

Listas. Operaciones con listas.

Crear una lista

La operación de creación de la lista admite distintos casos.
Se puede crear una lista vacía simplemente asignando valor nulo a su puntero inicial.
Se puede crear una lista agregando elementos al inicio de la lista, donde se enlaza el puntero al siguiente de cada nuevo nodo con primer nodo de la lista y luego se actualiza el puntero inicial con la posición del nuevo nodo.
Se puede crear una lista agregando elementos al final de la lista, donde se debe considerar el caso donde la lista esté vacía, para lo cual se actualiza el puntero inicial con la posición en memoria del primer nodo creado, o en el caso que ya se disponga de elementos en la lista, se debe recorrer la lista hasta ubicar el nodo final y actualizar el puntero al siguiente de este nodo con la posición en memoria del nodo que se agrega. Esta operación se puede optimizar si se lleva un registro del último nodo de la lista en un puntero auxiliar, lo cual permite no tener que recorrer toda la lista buscando el último nodo.
Por último se puede crear una lista agregando elementos con algún criterio de orden, donde se debe tener en cuenta el caso que la lista esté vacía, en donde se

actualiza el puntero inicial con el valor de la posición en memoria del nuevo nodo, o si ya se dispone de elementos en la lista, se debe recorrer la lista utilizando dos punteros auxiliares referenciando en un momento dado el nodo actual y su nodo anterior, hasta encontrar la posición en donde insertar el elemento de acuerdo al criterio establecido para la lista, y enlazar el nodo actual como nodo siguiente del nodo a insertar actualizando el puntero al siguiente del nuevo nodo con el valor de la posición en memoria del nodo actual y enlazar el nodo anterior al nuevo nodo actualizando su puntero al siguiente con el valor de la posición en memoria del nuevo nodo.
(código)

Recorrer una lista

En la operación de recorrido de una lista se recorre la lista completa desde el primero al último nodo. Se debe utilizar un puntero auxiliar para acceder a cada nodo inicializandolo con el valor del puntero inicial de la lista.
(código)

Buscar un elemento

En la operación de buscar un elemento se debe hacer un recorrido de la lista pero hasta llegar al elemento buscado, recuperando la posición del nodo en donde se encuentra el dato, o hasta llegar al final de la lista comprobando que el elemento no existe.
(código)

Borrar un elemento de la lista

En la operación de borrado de un elemento se debe tener en cuenta el caso que se deba borrar por posición y el caso que se deba borrar un elemento dado. En ambos casos se debe recorrer la lista hasta llegar a la posición a borrar, hasta encontrar el elemento a borrar o hasta llegar al final de la lista, comprobando que el nodo no existe o la posición no es válida.
Esta búsqueda se debe realizar utilizando dos punteros auxiliares apuntando al nodo actual y al nodo anterior para poder luego actualizar los enlaces quitando el nodo del elemento a borrar.

Una vez localizado el nodo a borrar se debe tener en cuenta el caso que ese nodo sea el primero de la lista, donde se actualiza el puntero inicial con la dirección en memoria del nodo siguiente. En el caso que el nodo a borrar se encuentre en una posición intermedia se debe actualizar el valor del puntero al siguiente del nodo anterior con el valor de la posición en memoria del nodo siguiente.

Por último en ambos casos se debe liberar el espacio ocupado en memoria por el nodo a borrar (dispose).
(código)

Agregar un elemento

En la operación de agregado de un elemento, se pueden diferenciar dos casos. Agregar un elemento al final de la lista o agregar un elemento al inicio de la lista. En el caso de agregar el elemento al final, se debe recorrer la lista para recuperar la posición del último nodo, o llevar un registro de la posición en memoria del último nodo en un puntero auxiliar que se debe actualizar con cada nodo que se agrega, y enlazar el nodo creado al último nodo actualizando el puntero al siguiente del último nodo con el valor de la posición en memoria del nuevo nodo.
En el caso de agregar el elemento al inicio, se debe enlazar el nuevo nodo con el primer nodo de la lista, actualizando el puntero al siguiente del nuevo nodo con el valor de la posición en memoria del primer nodo, valor dado por el puntero inicial y luego actualizar el puntero inicial con el valor de la posición en memoria del nuevo nodo.
En ambos casos si la lista esta vacía se actualiza el puntero inicial con el valor de la posición en memoria del primer nodo creado.
(código)

Insertar un elemento

En la operación de inserción de un elemento se debe tener en cuenta la posibilidad de insertar por algún criterio de orden o en una posición dada.

En ambos casos se debe tener en cuenta el caso que la lista esté vacía, donde se inserta el nuevo nodo simplemente actualizando el valor del puntero inicial con la dirección en memoria del nuevo nodo, o por el contrario se debe recorrer la lista desde el primer nodo y utilizando dos variables auxiliares que hagan referencia al nodo actual y su nodo anterior, hasta alcanzar la posición indicada recuperando así la dirección en la memoria del nodo en esa posición, o encontrando el nodo en la posición donde se cumpla con el criterio de orden, respectivamente.

Una vez que se dispone de la referencia donde se debe insertar el elemento se enlaza el nuevo nodo al nodo actual, actualizando el puntero al siguiente del nuevo nodo con el valor de la dirección en memoria del nodo actual, y enlazar el nodo anterior al nuevo nodo actualizando el valor del puntero al siguiente del nodo anterior con el valor de la dirección en memoria del nuevo nodo.
(código)

Acceder al k-ésimo elemento

En la operación de acceder a un elemento en una posición dada se debe recorrer la lista desde el primer nodo hasta alcanzar la posición, o sea que se recorre una cantidad de nodos igual al valor de la posición indicada recuperando la dirección en memoria del nodo en esa posición, o hasta llegar al final de la lista comprobando que la posición indicada no es una posición válida.

(código)

Buscar un elemento dado

En la operación de búsqueda de un elemento se debe disponer del dato a buscar para saber su dirección en memoria.
Se debe recorrer la lista desde el primer nodo hasta encontrar el nodo cuyo dato es el buscado y se recupera su posición en memoria, o hasta llegar al final de la lista comprobando que el dato no se encuentra en la lista.
(código)

Listas doblemente enlazadas y circulares. Características y operaciones.

En las ***listas circulares*** en el último nodo, en vez de almacenar un puntero nulo como siguiente elemento, se almacena la dirección del primer elemento de la lista, así desde un nodo cualquiera se puede acceder a cualquier otro nodo de la lista y las operaciones de concatenación y división son más eficientes. Tienen como desventaja que se pueden producir bucles o lazos infinitos.

Las ***listas doblemente enlazadas*** se pueden recorrer en ambos sentidos. Para esto cada nodo debe contar tanto con la dirección del nodo siguiente como la dirección del nodo anterior.

G - Recursividad

Características.

La ***recursividad*** es una herramienta que permite expresar la resolución de problemas evolutivos, donde es posible que un módulo de software se invoque a sí mismo en la solución del problema.
Se resuelve el problema por resolución de instancias más pequeñas del mismo problema y de la misma naturaleza.

Un subprograma recursivo es una función o procedimiento que se llama a sí mismo para resolver un problema, por resolución de sucesivas instancias mas pequeñas del mismo problema.

La reducción en el tamaño del problema garantiza llegar a un caso base que sería el caso final.

Ejecución de un programa y la pila de activación. Manejo de memoria en ejecución.

En la ejecución de un proceso cuando este llama a otro proceso que toma el control del procesador, este nuevo módulo se apila en memoria con sus instrucciones y datos.
Así funcionan los procesos recursivos y esta pila se la conoce como pila de activación, la cual tiene un tope y un fondo, donde cada elemento de la pila es código más datos y sólo pueden sacarse o desapilarse por el tope.

Análisis comparativo entre soluciones iterativas y recursivas. Ejemplos.

H - Concepto de Corrección. Análisis de algoritmos: concepto de eficiencia

Concepto de corrección.

Una vez que se ha escrito un programa se debe probar que sea ***correcto*** y un programa es correcto si se realiza de acuerdo a sus especificaciones, por lo cual es muy importante una completa especificación.

Técnicas para medir corrección.

Algunas técnicas que asisten al programador para medir la corrección de programas son el test, verificación, walkthrough o debugging que son usadas complementariamente para proveer evidencias para la corrección.

El ***testing*** es el proceso de proveer evidencias convincentes respecto de si el programa hace el trabajo esperado.
Para esto se debe diseñar un plan de pruebas, decidir cuáles aspectos del programa deben ser testeados y encontrar datos de prueba para cada uno de esos aspectos, se debe determinar el resultado que se espera que el programa produzca para cada caso de prueba, determinar casos límites, diseñar casos de prueba antes de escribir el programa en base a lo que el programa debe hacer.

Cuando se tiene el plan de pruebas y el programa el plan se aplica sistemáticamente, y cuando se descubre el error se localiza y se corrige.
Siempre que se corrige un error se debe probar el programa con el conjunto entero de pruebas.

El ***debugging*** es el proceso de descubrir y reparar la causa del error.

Esto puede involucrar el diseño y aplicación de pruebas adicionales para ubicar y conocer la naturaleza del error, y el agregado de sentencias adicionales en el programa para poder monitorear su comportamiento.

Los errores pueden provenir de un diseño defectuoso del programa, un algoritmo defectuoso que usa el programa, una codificación errónea del programa.

Algunas veces el error es evidente y se encuentra rápidamente el lugar del programa donde se encuentra la falla. Otras veces es necesario agregar sentencias de salida que sirven como puntos de control.

El ***walkthroughs*** es el proceso de recorrer el programa ante una audiencia.

La lectura de un programa a alguna otra persona, que no comparte preconceptos, provee un buen medio para detectar errores.

Verificación de Programas.

La ***verificación*** de un programa es el proceso de analizar las postcondiciones y en función de las precondiciones establecidas.

Las precondiciones junto con las postcondiciones permiten describir la función que realiza el programa sin especificar un algoritmo determinado.
Las precondiciones describen los aspectos que deben considerarse antes de que el programa se ejecute. Estas precondiciones son consideradas siempre verdaderas.
Las postcondiciones describen los aspectos que deben cumplirse cuando el programa terminó.

Concepto de eficiencia.

La ***eficiencia*** es una métrica de calidad que hace referencia a la capacidad de alcanzar un objetivo fijado haciendo un uso correcto de los recursos que se disponen.

Un algoritmo ***eficiente*** es aquel que realiza una administración correcta de los recursos del sistema en el cual se ejecuta.
Pensar en la optimización de un algoritmo requiere analizar el uso que éste hace de los recursos del sistema. La eficiencia.

Análisis de eficiencia de un algoritmo.

El ***análisis de la eficiencia*** de un algoritmo estudia el uso que éste hace de los todos los recursos como el ***tiempo*** que tarda el algoritmo en ejecutarse y la ***memoria*** que requiere.

Desde el punto de vista del tiempo de ejecución se consideran más eficientes aquellos algoritmos que cumplan con la especificación del problema en el menor tiempo posible. El recurso a optimizar es el tiempo de procesamiento.

Desde el punto de vista del uso de memoria serán eficientes aquellos algoritmos que utilicen las estructuras de datos adecuadas de manera de minimizar la memoria ocupada.

Análisis de algoritmos según su tiempo de ejecución y su utilización de memoria.

Para poder medir la eficiencia de un algoritmo desde el punto de vista de su tiempo de ejecución, hay que contar con una medida del trabajo que realiza para poder compararlos y seleccionar la mejor implementación.
Una medida es contabilizar la cantidad de operaciones realizadas. Así podemos tener algoritmos que siempre realizan una cantidad fija de operaciones y aquellos que la cantidad de operaciones se calcule en base a los datos, pudiendo distinguir entre un mejor y peor caso, aunque generalmente se tiene interés sobre el peor de los casos, del cual se obtiene una cota superior del tiempo de ejecución para cualquier entrada.
El tiempo de ejecución de un programa debe definirse en función de los datos de entrada.

Para poder medir la eficiencia de un algoritmo desde el punto de vista de la memoria utilizada se puede calcular únicamente la cantidad de memoria estática que utiliza el programa analizando las variables declaradas y el tipo correspondiente.

Análisis de Algoritmos: Análisis asintótico, comportamiento en el mejor caso, caso promedio y peor caso. Notación 0().

Se puede estimar el tiempo de ejecución de un algoritmo de dos formas.

Realizando un ***análisis teórico*** que busca obtener una medida del trabajo realizado por el algoritmo a fin de obtener una estimación teórica de su tiempo de ejecución, básicamente calculando el número de comparaciones y asignaciones.

Realizando un ***análisis empírico*** que se basa en la aplicación de juegos de datos diferentes a una implementación del algoritmo con el fin de medir sus tiempos de respuesta. Aplicando los mismos datos a distintas soluciones del mismo problema supone una herramienta de comparación entre ellas. Este método no tiene en cuenta la velocidad de la máquina o que los datos puedan favorecer más a una solución que a otra.

Por esto es bueno un análisis teórico que permita una comparación relativa.

Para esto:

> T(n) = O(F(n)) si existen constantes c y n_0 tales que T(n) >= c F(n) cuando n >= n_0

El tiempo de ejecución T(n) de un algoritmo se dice de orden F(n) cuando existe una función matemática F(n) que acota a T(n).
Así encontrando una función matemática que represente el tiempo de ejecución de un algoritmo se puede establecer un orden relativo entre las funciones.

Analizando así la función para un valor x de n, se puede establecer cual tomará más tiempo. También analizando el orden de crecimiento una función eventualmente puede ser mayor que otra o no dependiendo de los valores n.

Al decir que T(n) = O(F(n)) se esta garantizando que T(n) no crece más rápido que F(n), la cual es un límite superior para T(n).

Reglas generales de análisis

R1 - Para lazos incondicionales el tiempo de ejecución es a lo sumo el tiempo de ejecución de las sentencias que están dentro del lazo, incluyendo testeos, multiplicada por la cantidad de iteraciones que se realizan.

R2 - Para lazos incondicionales anidados el tiempo total de ejecución es el tiempo del bloque que contienen multiplicado por el producto del tamaño de todos los lazos.

R3 - Dado un fragmento `if(condición) then S1 else S2`, el tiempo no puede ser mayor al tiempo del testeo más el máximo entre los tiempos de S1 y S2.

R4 - Para sentencias consecutivas el tiempo es el máximo entre todas las sentencias.

Análisis de eficiencia en algoritmos recursivos.

La recursividad permite dar soluciones claras a problemas muy complejos pero tiene algunas desventajas desde el punto de vista de la eficiencia, debido a la sobrecarga de tiempo y memoria asociada a la llamada de subprogramas.
El tiempo de ejecución se puede calcular asociando la solución recursiva a una función de tiempo indefinida T(n) y desarrollada por partes.

```
       c + T(n - 1)   n > 1
T(n){
       d              n<= 1
```

Análisis de eficiencia en algoritmos de búsqueda y ordenación sobre vectores.

Para medir los diferentes algoritmos de ordenación es necesario calcular la cantidad de trabajo que realizan contando la cantidad de comparaciones e intercambios que realizan.

Peores casos:

Selección
C:= n (n - 1) / 2 I:= 3 (n - 1)
Burbujeo
C:= n (n - 1) / 2 I:= 3 n (n - 1) / 2
Inserción
n - 1 <= C <= n (n - 1) / 2
2 (n - 1) <= I <= 2 (n - 1) + n (n - 1) / 2

En el caso de ordenar las filas de una matriz por el valor de una columna en vez de mover físicamente las filas se suele usar un arreglo que indica según su orden como se deben leer las filas.

Métodos de ordenación eficientes.

Sorting by merge

Se divide el vector en dos sub vectores los cuales se ordenan por separado y luego se hace un merge de ambos.
A medida que se aumenta la cantidad de divisiones del vector original es más eficiente el proceso de ordenación. Esto concluye en un método recursivo.
Este algoritmo tiene la desventaja que para realizar el merge necesita una estructura de igual tamaño y si el vector es muy grande esto puede resultar prohibitivo.

Ordenación rápida

Es un método recursivo igual que el "sort by merge" pero que no utiliza merge, sino que ordena dos sub-vectores donde los elementos de la primer mitad son menores que los elementos de la segunda.
Este algoritmo en el peor de los casos resulta ser muy ineficiente.

Algoritmos numéricos y propagación de error.

J - Estructura de datos no lineales: árboles.

Introducción al concepto de datos no lineales.

Un árbol es un tipo de dato no lineal.

Terminología y definiciones básicas del tipo de dato árbol.

Un ***árbol*** es una estructura de datos jerárquica donde cada elemento se puede relacionar con cero o más elementos, los cuales llama hijos. Si el árbol no está vacío existe un único elemento que no tiene predecesor, es decir que no es hijo de ningún otro elemento, y que se lo conoce como raíz. Cualquier otro elemento del árbol tiene un único padre y es a su vez un descendiente de la raíz.

Un ***camino*** entre la raíz y un nodo hace referencia los nodos y sus enlaces por los que se pasa para llegar a ese nodo. Este es un camino único cuya ***longitud*** es la cantidad de aristas o enlaces entre la raíz y el nodo. Esto determina su ***profundidad*** y el ***nivel*** al cual pertenece el nodo.
Un ***nivel*** hace referencia a todos los nodo que posean la misma ***profundidad***.
La ***altura*** del árbol hace referencia al camino mas largo desde la raíz hasta una hoja del árbol.

Arboles binarios. Representación y operaciones.

Un ***árbol binario*** es aquel árbol donde cada elemento puede tener a lo sumo dos hijos.

```
type
     tipoElem: … ;
     arbolBinario: ^nodo;
     nodo: record
               elem: tipoElem;
               izq, der: arbolBinario;
          end;

var arbol: arbolBinario;
```

Árboles binarios de búsqueda. Representación y operaciones.

Un ***árbol binario de búsqueda*** es un árbol binario que se dice que esta ***ordenado*** porque existe algún orden sobre los datos que almacena de tal manera que siempre es posible recuperarlos en un orden dado. Así todo dato de un nodo del árbol es mayor que los datos almacenados en su subárbol izquierdo y menor o igual que los datos almacenados en su subárbol derecho.

Problemas que combinan árboles, listas y arreglos.

K - Tipos de datos abstractos.

Abstracción de datos.

Una de las tareas centrales en la informática, por la cual se puede caracterizar el mundo real para poder resolver problemas concretos mediante el empleo de herramientas informáticas. Esto significa reconocer los objetos del mundo real y ***abstraer*** sus aspectos fundamentales y su comportamiento, de modo de representarlos sobre una computadora.

Conceptos sobre tipos de datos.

Todas las nociones de tipos, estructuras de datos, variables y constantes son abstracciones son abstracciones para tratar de acercar la especificación de los datos de problemas concretos al mundo real.

El concepto de ***tipo de dato*** es una necesidad de los lenguajes de programación que conduce a identificar valores y operaciones posibles para variables y expresiones.

Módulos, interfaz e implementación.

Al modularizar un sistema de software en procedimientos y funciones, se logra abstraer las operaciones de modo de descomponer funcionalmente un problema.

Un módulo se puede ver como una caja negra con una función interna y una interfaz de vinculación con otros módulos. Esta interfaz se la conoce como la parte pública del módulo, mientras que la implementación es la parte privada.

Encapsulamiento de datos.

Un paso tendiente a la abstracción de datos es lograr ***encapsulamiento*** de los datos, definiendo un nuevo tipo e integrando en un módulo todas las operaciones que se pueden hacer con él.

Si el lenguaje permite separar la interfaz de la implementación se tendrá ocultamiento de datos.

Si la solución del cuerpo del módulo puede modificar la representación del objeto-dato, sin cambiar la interfaz del módulo, se tendrá independencia de la representación.

Diferencia entre tipo de dato y tipo abstracto de dato.

El concepto de ***tipo abstracto de dato*** es un tipo de dato definido por el programador que especifica la representación de los elementos del tipo, las operaciones permitidas con el tipo y debe permitir encapsular la implementación de esas especificaciones.

La forma de programación que se corresponde con la ecuación de Wirth es

```
Programa = Datos + Algoritmos
```

Dado un problema a resolver se busca representar los objetos que participan en el mismo e incorporar los algoritmos necesarios para llegar a la solución. De esta forma, el programa implementado se aplica únicamente a la solución del problema original ya que funciona como un todo.

Esto se puede mejorar expresando la sección de algoritmos como

```
Algoritmos = Algoritmos de datos + Algoritmos de control
```

donde se entiende como Algoritmos de datos a la parte del algoritmo encargada de manipular las estructuras de datos del problema, y Algoritmos de control a la parte que representa el método de solución del problema, independiente de las estructuras de datos seleccionadas.

Dado que un TAD en su definición reúne la representación y el comportamiento de los objetos del mundo real se puede escribir la ecuación inicial como

```
Programa = TAD + Algoritmos de Control
```

que describe el enfoque de desarrollo utilizando Tipos Abstractos de Datos.

El uso de TADs tiene sus ventajas, ya que lleva en sí mismo la representación y el comportamiento de sus objetos lo cual los hace independiente del programa o

módulo que los utiliza permitiendo desarrollar y verificar su código de manera aislada. Esto facilita la reusabilidad del código.
El programa que referencia al TAD lo utiliza como una caja negra de la cual se obtienen resultados a través de operaciones predefinidas.

Al momento de diseñar y desarrollar un TAD no interesa conocer la aplicación que lo utilizará.
Al momento de utilizar un TAD no interesa saber como funciona internamente.

Requerimientos y diseño de TADs.

Disponer de un TAD brinda la posibilidad de tener código reusable, donde se representa la estructura de datos y su comportamiento.
Esto requiere poder encapsular dentro de un módulo la especificación y la implementación de las operaciones, poder declarar tipos protegidos donde la representación esté oculta de la parte visible del módulo y poder crear instancias a partir de ese molde.

El diseño de un tipo de dato abstracto lleva consigo la selección de una representación interna seleccionando las estructuras de datos adecuadas para representar la información y las operaciones a proveer para ese tipo las cuales pueden ser de creación/inicialización, modificación o de análisis de los objetos del tad.

Ejemplos TAD pila, TAD cola.
(Buscar en diapositivas)

L - Introducción a la Programación Orientada a Objetos.

Motivación. Reusabilidad de soluciones.

Abstracción de datos y procesos.

La noción de Objeto. Operaciones (métodos) aplicables a un objeto.

Concepto de clases e instancia.

Noción de herencia. Relación con el re-uso.

Aplicaciones.

Caracteristicas de 10s lenguajes enfocados a POO.

M - Conceptos iniciales de concurrencia

Motivación: arquitecturas de computadoras actuales, aprovechamiento de los procesadores.

Definiciones.

Ejemplos.