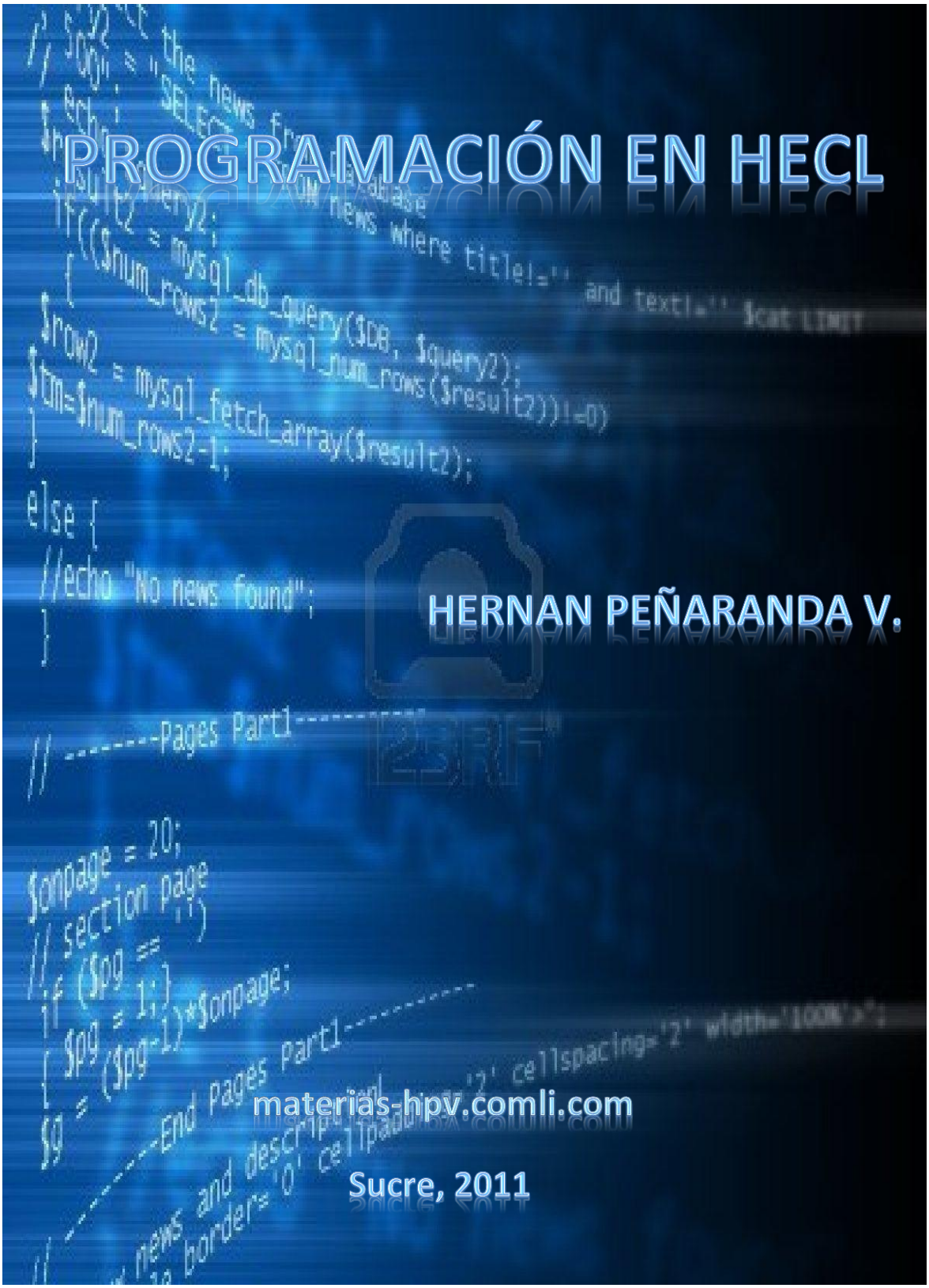


PROGRAMACIÓN EN HECL

HERNAN PEÑARANDA V.

materias-hpv.comli.com

Sucre, 2011

1. LA PROGRAMACIÓN ESTRUCTURADA Y SU IMPLEMENTACIÓN

1.1 Introducción

Durante el transcurso de esta materia aplicaremos los principios de la programación estructurada, en las fases de análisis, diseño e implementación de los diferentes programas que elaboremos. El objetivo es que al finalizar el curso estén habituados a aplicar los principios de la programación estructurada en todas las fases de la creación del software.

1.2 Estructura y lógica de los primeros programas

En los primeros días de la computación las computadoras contaban con muy limitados recursos de procesamiento y memoria: velocidades de procesamientos inferiores a 1 MHz y memoria nunca superior a los 64 kb. Por ello los primeros programas eran muy pequeños y la lógica de los mismos se centraba en optimizar tanto la velocidad (capacidad de procesamiento) como el tamaño de los programas (memoria disponible). No era muy importante el que los programas fueran entendibles, porque en general era la misma persona la que programaba, mantenía y modificaba el programa.

Los problemas comenzaron a surgir cuando por algún motivo el programador dejaba de estar disponible para mantener el programa. Entonces por lo general el programador contratado para mantener el programa se veía forzado a crear su propio programa, porque la lógica del programa anterior resultaba extremadamente difícil de comprender. Dicha lógica era parecida a la que se muestra en el siguiente diagrama:

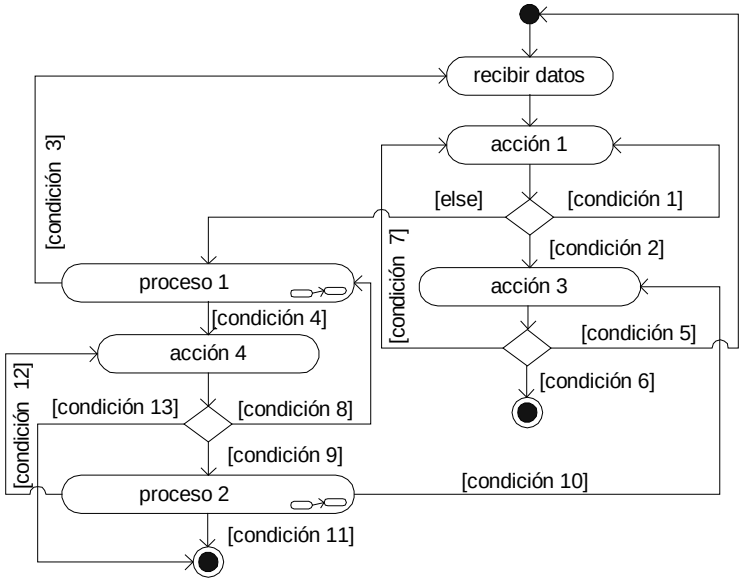


Figura 1.1. Lógica común en un programa no estructurado

A medida que se incrementaban las capacidades de procesamiento y de memoria, los problemas de mantenimiento también crecían, incrementando considerablemente el costo del software, porque los programas debían ser reescritos una y otra vez.

1.3 La programación estructurada

Fue bajo esas condiciones que surgió, en los años 60, la ***programación estructurada (PE)***, en base a las prácticas y métodos que ya empleaban algunos

programas y que habían comprobado su eficacia en la elaboración y mantenimiento de programas. La programación estructurada formó parte de las llamadas *técnicas para el mejoramiento de la productividad en programación.*

Su propósito era el de facilitar la creación y mantenimiento de los programas: si todos los programadores crean sus programas basados en los mismos principios, es de suponer que los programas elaborados sean entendidos por todos, sin importar quién sea el creador del programa.

Esa motivación llevó a que esta metodología fuera adoptada por la mayoría de la comunidad informática y llegó a ser la metodología de programación que dominó el desarrollo de aplicaciones en el ámbito de la informática durante los años 70 y 80. Actualmente ya no ocupa ese lugar privilegiado, pero los fundamentos que dieron lugar a su origen y que se aplicaron con éxito durante décadas, siguen siendo tan útiles y válidos como cuando fue creada. En realidad dichos principios forman parte de las metodologías de desarrollo actuales, por lo que el estudio de la programación estructurada no sólo resulta de utilidad práctica sino que es imprescindible si se para adquirir buenos hábitos en la labor de desarrollar software.

1.4 Fundamentos De La Programación Estructurada

Como ya se mencionó previamente, la programación estructurada está constituida por los principios, técnicas y métodos que demostraron su eficacia en la creación, mantenimiento y actualización del software. Dichos principios son los siguientes:

a) *Dividir un problema complejo en problemas más sencillos (programación descendente).*

b) *Emplear estructuras estándar para construir la totalidad del programa*.

c) *Emplear tipos de datos a la medida.*

El primer principio es el más importante al momento de hacer el análisis del problema. Básicamente nos dice que debemos analizar el problema y dividir el mismo en problemas más sencillos y si estos son todavía complejos, dividirlos más hasta que cada problema sea lo suficientemente sencillo como para poder ser resuelto sin mayor dificultad y de manera independiente. Esto último es importante, porque no se trata sólo de dividir el problema, sino hacerlo de forma tal que cada uno de los problemas resultantes pueda ser dividido de forma independiente. A cada uno de los problemas resultantes de esta división se le da el nombre de módulo y la característica más importante de un módulo es justamente esa, puede ser analizado y resuelto de manera independiente, sin importar que otros módulos estén involucrados en el sistema. Por lo anterior, a este principio se le conoce también con el nombre de programación modular y debido a que la división del problema (en otros más pequeños) se hace de arriba (del problema más complicado) hacia abajo (a los problemas más simples), se conoce también como programación descendente (Top-Down en inglés).

El segundo principio nos dice que debemos emplear sólo tres estructuras estándar para construir la totalidad del programa: la secuencia, la selección y la iteración. Este principio fue propuesto por dos matemáticos: Corrado Böhm y Guiseppe Jacopini y fue demostrado por los mismos. Actualmente, sin embargo, se puede considera que una estructura es estándar si tiene una sola entrada y una sola salida y ese es el principio que emplearemos en esta materia. El que el programa sea estructurado, nos permite armar un programa colocando una estructura a continuación de otra (pues como ya se dijo cada estructura sólo debe tener una entrada y una salida), de esta manera el programa puede ser leído y la lógica comprendida fácilmente.

El tercer principio de la *programación estructurada* señala que debemos definir con claridad y precisión los tipos de datos que se requieren, se usan y devuelven desde un módulo. Actualmente la mayoría de los lenguajes de programación son muy flexibles en este aspecto, sin embargo, no ocurre lo mismo con la lógica y aunque el lenguaje no controle este aspecto el programador debe hacerlo, por lo que este principio, aunque no esté presente en el lenguaje, siempre debe estar presente en la lógica de un programa.

1.5 Programación Estructurada y la Programación Orientada a Objetos

La metodología de programación que pasó a ocupar el lugar de la programación estructurada (desde finales de la década de los ochenta), ha sido la Programación Orientada a Objetos (POO). En esta metodología, todos los elementos con los que se trabaja en un programa (o sistema) son objetos. Un objeto en un trozo independiente de software que no sólo tiene datos, sino también las funciones que operan sobre dichos datos, es decir no sólo conoce con qué datos trabajar sino que además sabe cómo trabajar con esos datos.

Prácticamente la diferencia entre la Programación Estructurada (PE) y la POO, es que en la PE, por un lado se tienen los datos, en forma de estructuras de datos, y por otro se tienen las funciones que operan sobre dichos datos (los procedimientos y/o funciones), mientras que en la POO, ambos elementos se encuentran en un solo bloque denominado Objeto.

No se trata sólo, como podría pensarse, de formas diferentes de organizar los elementos de un programa, sino principalmente de formas diferentes de pensar, de analizar los problemas. Por ejemplo, si se quiere elaborar un software para dibujar figuras geométricas, en el análisis estructurado, pensaríamos por un lado en las funciones que se requieren: dibujar punto, dibujar línea, dibujar triángulo, dibujar rectángulo, dibujar círculo, etc., y por otro pensaríamos en las estructuras de datos que son necesarias para esas funciones: puntos, listas de puntos, estructuras de color, etc. En el análisis orientado a objetos, por el contrario, pensaríamos en los objetos (o clases) que se requieren para el sistema: objeto punto, objeto línea, objeto triángulo, objeto círculo, etc., y para cada uno de dichos objetos pensaríamos tanto en los datos como en las funciones que deben operan sobre dichos datos.

Entonces, cuando analizamos un problema desde el punto de vista estructurado pensamos en las funcionalidades que debe tener el sistema, mientras que cuando analizamos un problema desde el punto de vista orientado a objetos pensamos en los objetos que deben integrar el sistema.

Cuando hemos resuelto un problema siguiendo los principios de la PE, empleamos dicha solución llamando a las funciones con los datos (o estructuras de datos) que las mismas requieren. Las funciones por su parte devuelven las respuestas también en forma de datos o estructuras de datos. Así por ejemplo si se ha elaborado un software para mostrar elementos gráficos en pantalla (como las de Windows), llamaríamos a dicho software de la siguiente forma Create_Window (Mi_Ventana); Create_Label(Mi_Etiqueta). Donde las funciones son “Create_Window” y “Create_Label”, y los datos son “Mi_Ventana” (con los datos de la ventana: tamaño, color, posición, etc.) y “Mi_Etiqueta” (con los datos de la etiqueta: título, color, posición, etc.).

Cuando hemos resulto un problema siguiendo los principios de la POO, empleamos dicha solución llamando a los objetos tanto para que cambien sus datos como para que lleven a cabo alguna actividad o devuelvan algún resultado. Así, para el caso anterior: un software que muestra elementos gráficos en la pantalla, llamaríamos a dicho software de la siguiente forma: Desktop.Window.Create, Desktop.Window.Title(‘Mi Título’), Desktop.Label.Create,

Desktop.Label.Caption('Mi Texto'), etc. Donde "Desktop" es el nombre del software, "Window", "Label", etc. son algunos de los objetos de dicho software, "Create" es una de las funciones de los objetos y "Title", "Caption", son algunas de las funciones que permiten cambiar los datos de los objetos.

Una de las razones por las cuales la POO, ha ganado popularidad es porque ha sido asociada erróneamente con los "objetos" gráficos que aparecen en sistemas operativos como Windows (ventanas, iconos, botones, etc.), cuando en realidad dichos "objetos" son el resultado de llamadas a "funciones" del sistema operativo, en consecuencia no son propiamente "objetos", sino funciones. Un aspecto importante que se debe tomar en cuenta al momento de desarrollar un software es que todos los sistemas operativos, incluido Windows 7 y las últimas versiones de Linux, han sido creados con los principios de la PE, no de la POO. Entonces cabe preguntarse: Si la POO es superior a la PE, ¿por qué todos los sistemas operativos actuales son estructurados y no orientados a objetos?

Se debe señalar además que actualmente existen otras metodologías, como la PGA (Programación Guiada por Agentes), que conceptualmente son una evolución de la POO y que en consecuencia podrían desplazar a la misma en un futuro no muy lejano.

1.6 Implementación de la Programación Estructurada

Actualmente tenemos a nuestra disposición una gran variedad de lenguajes de programación que pueden ser empleados para implementar en la práctica los principios de la programación estructurada. De una u otra forma, la mayoría de dichos lenguajes han sido influenciados por la POO, por lo que actualmente es muy difícil acceder a un lenguaje puramente estructurado.

El lenguaje estructurado por excelencia es Pascal (el cual ha sido creado específicamente para enseñar los principios de la PE), sin embargo, las versiones actuales de dicho lenguaje también han recibido la influencia de la POO y tanto en las versiones libres como pagadas, lo que tenemos es lo que se ha venido a denominar "Object Pascal" una "evolución" del Pascal hacia la POO.

Lamentablemente no contamos en la Carrera con computadoras suficientes como para impartir la materia en dichas herramientas (como debería ser). Ahora bien, si no se aplican en la práctica los principios de la programación estructurada, elaborando y haciendo correr programas, esto es si la enseñanza es solamente teórica, el aprender de memoria dichos principios es apenas si de alguna utilidad.

Si bien los costos de las computadoras portátiles han disminuido considerablemente, la mayoría de los estudiantes no cuentan aún con una de ella y no se les puede exigir que las adquieran porque en muchos casos están fuera de sus presupuestos.

Por ello nos vemos forzados a recurrir a otros medios no convencionales, que aunque menos adecuados para este fin, constituyen de hecho una alternativa muy superior a la del aprendizaje meramente teórico. En ese sentido, y tomando en cuenta que la mayoría de los estudiantes cuentan con un teléfono celular, con la tecnología Java corriendo en ellos, se ha optado por utilizar dichos dispositivos en la enseñanza de la materia.

Si bien los celulares han sido creados con propósitos muy diferentes a los de la enseñanza, estos dispositivos son en realidad mini-computadoras, con capacidades de procesamiento y memoria muy superiores a las de las primeras computadoras y a las computadoras personales de hace algunos años, por

lo que tienen el potencial como para ser empleados en la enseñanza de la materia.

En ese sentido existen ya algunos avances y actualmente se cuenta con unas cuantas herramientas disponibles para este fin y es de esperar que la disponibilidad de estas herramientas incremente en el futuro. De entre ellas podemos mencionar a *MobilBasic*, una aplicación que permite programar en Java, lamentablemente se trata de una aplicación comercial y ese hecho hace que no sea la mejor alternativa. *Cy4th*, una aplicación que permite ejecutar instrucciones y elaborar programas en lenguaje Forth, se trata de una aplicación gratuita, por lo que se constituye en una alternativa plausible para la enseñanza de la materia. *MathPro*, es una aplicación pensada sobre todo en la solución de problemas numéricos, su interfaz de desarrollo es bastante avanzada para ser una aplicación que corre en celulares, cuenta con un lenguaje que si bien sólo tiene un conjunto reducido de instrucciones, son suficientes para resolver prácticamente cualquier problema, su principal inconveniente (al igual que *MobilBasic)* es que se trata de una aplicación comercial. *Hecl*, es un lenguaje de programación que corre tanto en celulares como en computadoras convencionales, cuenta con la mayoría de las estructuras de programación y es una herramienta gratuita, en realidad no solo gratuita, sino que es libre, por lo que no solo puede ser empleada sin costo, sino que además es posible modificar la aplicación en si pues nos proporcionan el código fuente de la misma.

Por las características antes descritas se ha optado emplear como herramienta y lenguaje de programación para la enseñanza de la materia “hecl”. No obstante, cuando sea conveniente se recurrirá también a otros lenguajes.

1.7 Introducción a hecl

Tanto los ejecutables (.jar), como el código fuente, así como la ayuda para este lenguaje se encuentra disponible en http://www.hecl.org/

Para instalar la aplicación en un celular que tenga implementada la tecnología Java, y la mayoría de los celulares la tienen, sólo se requiere el archivo “Hecl.jar” que ocupa unos 159 Kb (la versión midp2.0) para celulares relativamente recientes (con unos 5 años de antigüedad) y unos 75 kb (la versión midp1.0) para celulares más antiguos. También existen versiones específicas para celulares Blackberry y mwt. Todos estos archivos se encuentran en el directorio “jars” dentro de la carpeta “hecl” que es la carpeta que se obtiene cuando se baja y descomprime la aplicación desde internet. A propósito cuando baje hecl, se recomienda descomprimir el archivo resultante con 7-zip, que también es un descompresor gratuito (http://wwww.7-zip.org), tome en cuenta que para obtener la carpeta “hecl” es necesario descomprimir el archivo dos veces.

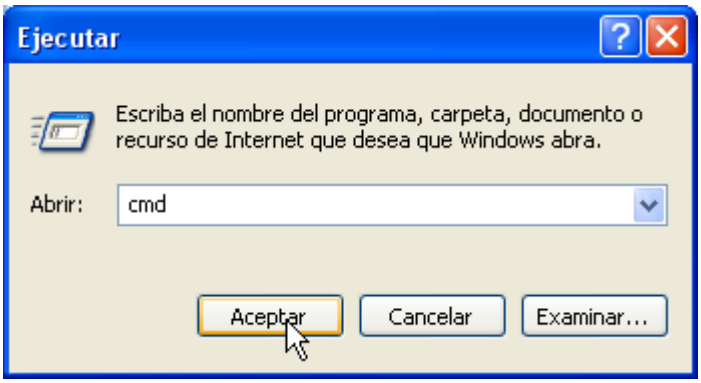
La aplicación “hecl” para celulares se instala igual que cualquier otra aplicación “.jar”, es decir se copia el archivo hecl.jar al celular y una vez en el celular se procede a su instalación. Algunos modelos requieren además del archivo “hecl.jar” el archivo “hecl.jad”, que también se encuentra en los directorios antes mencionados.

Para hacer correr “hecl” en una computadora se requiere el archivo “Hecl.jar” que se encuentra en el directorio “J2SE” (dentro del directorio “Jars”). Además es necesario que en la computadora esté instalado Java, como mínimo la versión 1.5, aunque se recomienda la versión 1.6. Si no está instalado java deberá bajar el “jdk-6” o posterior del sitio de Sun (http://java.sun.com).

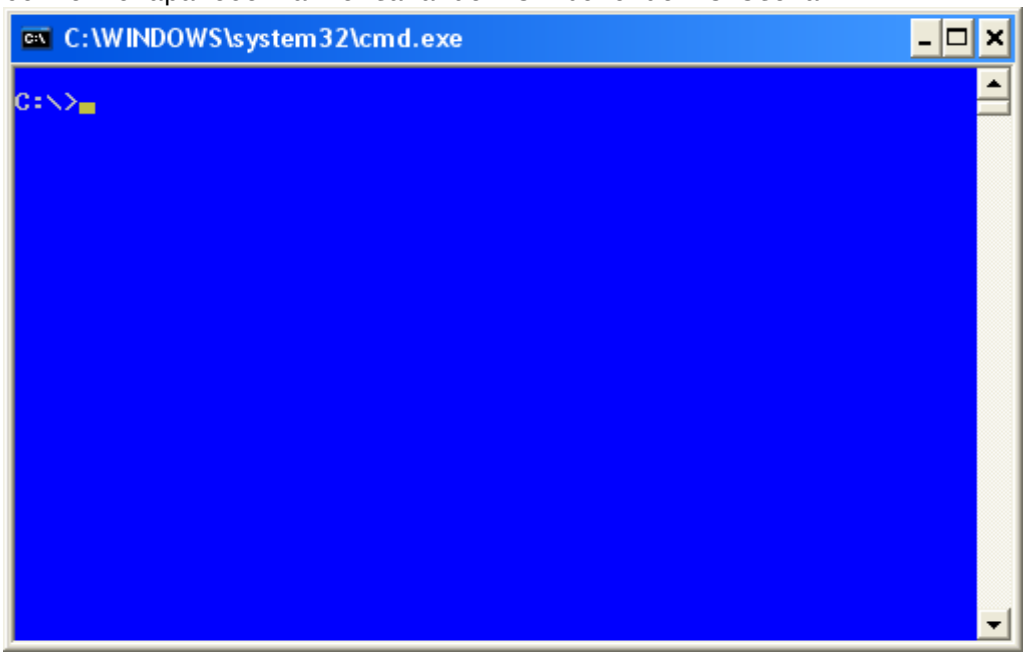
Para trabajar con “hecl” se debe trabajar en la línea de comandos de Windows, también conocido como “Símbolo del sistema”. Para acceder el “símbolo

del sistema", simplemente se hace correr el programa "Ejecutar" (que se encuentra en el menú inicio de windows, aproximadamente como se muestra en la siguiente figura:

Y en la ventana que aparece se escribe la instrucción "cmd" y se hace clic en el botón "Aceptar", tal como se muestra en la siguiente figura:



Con ello aparece la ventana del símbolo del sistema:



Para hacer correr “hecl” se debe escribir la siguiente instrucción:

java –jar <camino y nombre del archivo hecl.jar>

Por ejemplo, si el archivo “hecl.jar” se encuentra en el directorio: e:\hecl\jars\j2se\, escribimos:

```
C:\>java -jar e:\hecl\jars\j2se\hecl.jar
hecl>
```

Y como se ve, el símbolo del sistema cambia a “hecl>” mostrándonos que ahora estamos ya dentro de “hecl” (se reitera que para que esta instrucción funcione es necesario que Java esté instalado en la computadora).

Una vez que hecl está corriendo podemos ejecutar las instrucciones del lenguaje. Así, como ya es tradicional en todos los lenguajes, lo primero que podemos hacer es mostrar el mensaje “Hola Mundo” en pantalla.

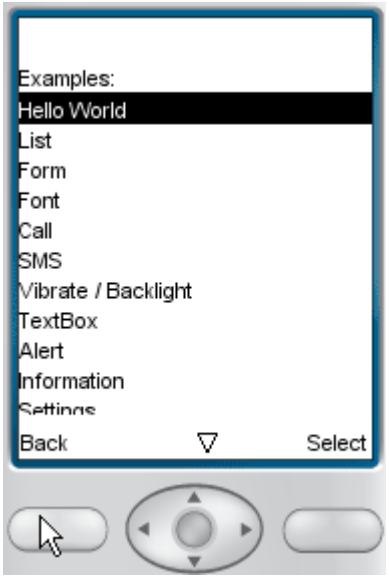
Antes, debemos hacer notar que hecl es sensitivo a mayúsculas y minúsculas, es decir que las instrucciones deben ser escritas tal y como está especificado en el lenguaje sin cambiar nada a mayúsculas o a minúsculas. A propósito casi todas las instrucciones en hecl están en minúsculas.

Para ejecutar un comando en hecl, simplemente se escribe el nombre del comando y a continuación él o los parámetros (o datos) separados con espacios. En hecl (a diferencia de otros lenguajes) no se emplean paréntesis. Así por ejemplo, el comando que nos permite mostrar un mensaje (o un resultado) en pantalla es “puts”, que recibe un solo parámetro, por lo tanto para mostrar el mensaje hola mundo escribimos: puts “Hola Mundo”:

```
hecl> puts "Hola Mundo"
Hola Mundo
hecl>
```

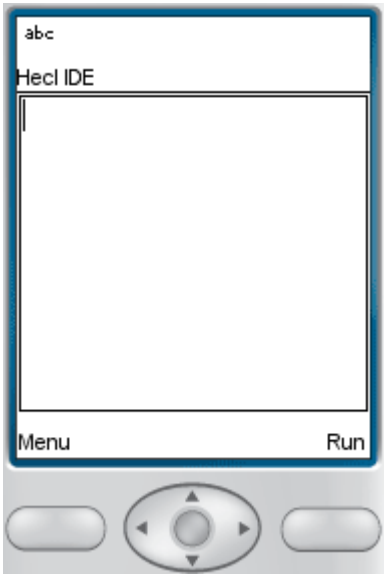
Y como se ve aparece dicho mensaje en pantalla. De esa manera hemos ejecutado nuestra primera orden en hecl y como se ha podido comprobar no es necesario elaborar un programa para ello.

Lo mismo es válido cuando se hace correr hecl en el celular. En este caso, si se hace correr la versión para MIDP2, aparece la siguiente pantalla:



Donde el color y el número de opciones visibles según el modelo de celular que se esté empleando. Para comenzar a escribir las instrucciones se presiona el botón correspondiente a la opción “Back”, y luego otra vez el

mismo botón o la tecla de borrado “C” (dependiendo del celular). Las instrucciones (y programas) se escriben en la ventana resultante:



La única diferencia, con relación al símbolo del sistema, es que en el celular, después de escribir la orden, se debe pulsar el botón correspondiente a la palabra “Run” para que la orden sea ejecutada:

El resultado de la instrucción aparece en otra ventana, tal como se muestra en la figura de la siguiente página. Para seguir escribiendo instrucciones o corregir algún error cometido, simplemente se pulsa el botón correspondiente a la palabra “OK”.

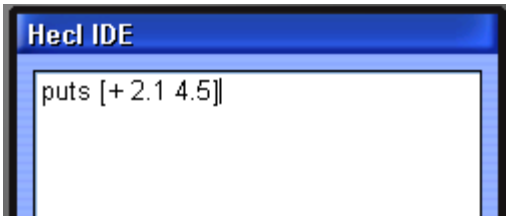
Continuemos ahora viendo algunas otras instrucciones en hecl, antes sin embargo, debemos aclarar que en hecl las comillas se emplean para agrupar datos, de manera que sean tratados como uno sólo, no para diferenciar texto de otros tipos da datos (como ocurre en otros lenguajes). Por ejemplo si en lugar de mostrar “Hola Mundo” queremos mostrar el mensaje “Hola_Mundo”, que es un solo dato, la instrucción no requiere comillas, tal como se muestra en la siguiente instrucción:


```
hecl> puts Hola_Mundo
Hola_Mundo
hecl>
```

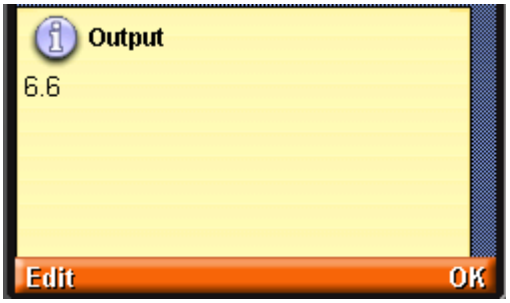
Quizá el aspecto al que más cuesta acostumbrarse cuando se trabaja con hecl es el relativo a los operadores tanto matemáticos como lógicos. En hecl primero se escribe el operador y después él o los datos con los que trabaja dicho operador. Por ejemplo para sumar 2.1+4.5 escribimos:

```
hecl> + 2.1 4.5
6.6
hecl>
```

Si se trabaja con el celular, la anterior orden no produce ningún resultado visible. Para ver los resultados en el celular se debe emplear el comando “puts” y como dicho comando sólo acepta un parámetro, es necesario encerrar la operación entre corchetes. Los corchetes en hecl operan de manera similar a los paréntesis en otros lenguajes: las instrucciones escritas dentro de corchetes se ejecutan primero. Entonces para ejecutar la anterior orden en un celular se debe escribir puts [+ 2.1 4.5]:



Ahora al ejecutar la orden (Run), se obtiene el resultado buscado:



En hecl podemos relizar las cuatro operaciones aritméticas básicas (suma: +, resta: -, multiplicación: *, división: /) tanto con números enteros como con números reales. Por ejemplo podemos dividir 3.4/6.7:

```
hecl> / 3.4 6.7
0.5074626865671642
hecl>
```

Dichos operadores funcionan tanto con números reales como con números enteros. El que los datos sean tratados como números reales o enteros depende de los datos que se manden al operador. Así si los dos datos son enteros, las operaciones se realizan con aritmética de números enteros. Por ejemplo el resultado de dividir 7/4 (números enteros) es:

```
hecl> / 7 4
1
hecl>
```

Que como vemos es el cociente de la división entera. En este caso, para obtener el residuo empleamos el operador %:

```
hecl> % 7 4
3
hecl>
```

Si uno de los datos es real o si todos los datos son reales, las operaciones se realizan con aritmética de punto flotante, por ejemplo si dividimos 7./4 obtenemos:

```
hecl> / 7. 4
1.75
hecl>
```

Que como vemos es el resultado real. Algunos operadores, como los de de suma, multiplicación y resta aceptan 3 o más argumentos. Así por ejemplo para sumar 3+4+5+6+7 escribimos:

```
hecl> + 3 4 5 6 7
25
```

Igualmente para multiplicar 1*2*3*4*5*6*7 (factorial de 7) escribimos:

```
hecl> * 1 2 3 4 5 6 7
5040
```

Y para restar (((23-5)-4)-8)-3 escribimos:

```
hecl> - 23 5 4 8 3
3
```

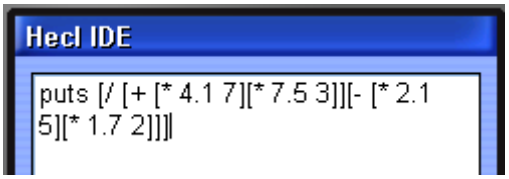
Para evaluar expresiones más complejas debemos recurrir a los corchetes, que como ya dijimos permiten agrupar expresiones. Así por ejemplo para calcular el resultado de la siguiente expresión:

$$\frac{4.1*7+7.5*3}{2.1*5-1.7*2}$$

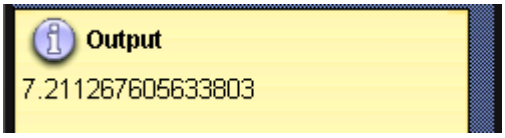
Escribimos:

```
hecl> / [+ [* 4.1 7][* 7.5 3]][- [* 2.1 5][* 1.7 2]]
7.211267605633803
```

Recuerde que si está trabajando con el celular, debe además emplear el comando puts:



Y por supuesto se obtiene el mismo resultado:



Hecl cuenta con la mayoría de las funciones matemáticas de uso más frecuente, sin embargo, en la versión del celular, el conjunto de funciones disponible es reducido. Así en el celular se tienen las funciones trigonométricas directas: sin, cos, tan, pero no las inversas (asin, acos, atan), ni las hiperbólicas (sinh, cosh, tanh). Tampoco están disponibles las funciones logarítmicas (log y log10), las exponenciales (exp) y las de potencia (pow). Están disponibles las funciones ceil, floor e int, pero no round (redondeo) ni signum (signo). Está disponible igualmente sqrt (raíz cuadrada) pero no cbrt (raíz cúbica). Están disponibles las funciones de conversión toDegrees y toRadians.

1.8 Ejemplos

Para adquirir práctica en hecl, calcularemos el valor de las siguientes expresiones:

1. $\sqrt{\frac{\sin(9.2)-\cos(8.32)}{\tan(6.89)}}$

```
hecl> sqrt [/ [- [sin 9.2][cos 8.32]][tan 6.89]]
0.9840519243460468
```

2. $\frac{|\sin(2.2)-\cos(3.4)|}{\tan(0.9)}$

```
hecl> / [abs [- [sin 2.2][cos 3.4]]][tan 0.9]
1.4087870647307301
```

3. $\sqrt{\frac{\sin(45°)-\cos(70°)}{3.4\tan(80°)}}$

```
hecl> sqrt [/ [- [sin [toRadians 45]][cos [toRadians 70]]][*
 3.4 [tan [toRadians 80]]]]
0.13759982959015346
```

4. Parte entera de: $\sqrt{14.2}-\sqrt{3.2}$

```
hecl> int [- [sqrt 14.2][sqrt 3.2]]
1
```

5. Entero más pequeño mayor o igual al resultado de: $6.8*\sin(125°)$

```
hecl> ceil [* 6.8 [sin [toRadians 125]]]
6.0
```

6. Resíduo de la división: $\frac{(6+5)*5}{3+4}$

```
hecl> % [* [+ 6 5] 5][+ 3 4]
6
```

7. $\sqrt{\frac{(3+4.1)-1.1*3.2}{6.53+\cos(6.7°)}}$

```
hecl> sqrt [/ [- [+ 3 4.1][* 1.1 3.2]][+ 6.53 [cos [toRa]]]
dians 6.7]]]]
0.6898283787288538
```

Ejercicios

Calcule el valor de las siguientes expresiones:

1. $\sqrt{\frac{6.75^2}{6.2+3^3}}\cos(30.2°)$

2. $sin\left(\frac{\sqrt{7.2}+9.67}{8.43}\right)$

3. $\sqrt{\frac{5^3-2^2+3}{(7-3)}}$

4. Entero más pequeño mayor o igual al resultado de: $\left(\sin(9.2)-\frac{\cos(5.6)}{\tan(9.2)}\right)$

5. Entero más pequeño menor o igual al resultado: $\left(\frac{4.5^2-9.8^3}{\sqrt{6.32+2^3}}\right)$

6. Cociente de la división: $\frac{5!+3^4}{4^3}$

7. $\frac{\sin(0.35)+\cos(0.89)+\tan(5.45)}{3!*4^5}$

8. $\frac{4.5^3+9.8^2+2.14^4}{\sqrt{6.32+0.98^2}}$

2. SECUENCIA y MÓDULOS

En este capítulo comenzaremos el estudio de la primera estructuras estándar: *la secuencia*. El conocimiento y aplicación de esta estructura nos ayudará a aplicar el segundo principio de la programación estructurada: *emplear estructuras estándar para construir la totalidad del programa.* Sin embargo, y desde un principio, aplicaremos también el primer principio de la programación estructurada: la programación modular.

El objetivo del presente capítulo es que al concluir el mismo estén capacitados para resolver problemas simples empleando la secuencia y la programación modular.

No debemos olvidar que el propósito de los principios de la programación estructurada es el lograr un estilo de programación claro y ordenado, de manera que facilite la creación y mantenimiento del software.

2.1 Diagramas de actividades

Antes de comenzar el estudio de la secuencia explicaremos la simbología que emplearemos en la elaboración de los algoritmos: los *diagramas de actividades.*

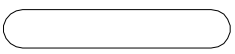
Los algoritmos describen la secuencia lógica de acciones que deben llevarse a cabo para resolver un determinado problema. Existen muchas formas en las que se pueden elaborar algoritmos, de ellas emplearemos en esta materia los *diagramas de actividades*.

Los *diagramas de actividades* constituyen una de las varias clases de diagramas que forman parte de *UML*, el *Lenguaje de Modelado Unificado*, que actualmente es el lenguaje estándar para el modelado de sistemas de software. Se ha elegido este tipo de diagramas no sólo por ser un estándar, sino porque permiten expresar los algoritmos de manera clara, ordenada, sencilla y porque además son muy versátiles, permitiendo representar diferentes tipos de estructuras.

El inicio de un diagrama de actividades se representa con un círculo relleno:

El final del diagrama de actividades se representa con un círculo que contiene un círculo relleno en su interior:

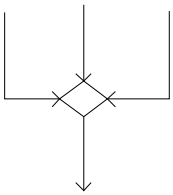
Una acción (o sentencia) se representa con un rectángulo con sus bordes redondeados:



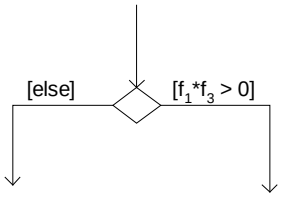
Un flujo, que es el símbolo empleado para unir los elementos del diagrama y señalar la dirección en que se deben seguir las acciones, se representa por una flecha continua abierta:

Una unión o bifurcación se representa por un pequeño rombo:

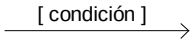
Cuando se emplea como unión llegan dos o más flujos y sale uno.



Cuando se emplea como bifurcación ingresan uno o más flujos y salen dos o más flujos acompañados de alguna condición.



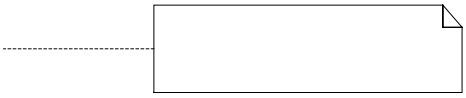
Una condición, tal como se puede observar en la anterior figura, se representa como texto encerrado entre corchetes y siempre está relacionado a algún flujo:



El texto de la condición puede ser una expresión matemática, una expresión relacional, una condición textual, etc. Para la condición por defecto se puede emplear *[else]* (caso contrario).

Una actividad representa un conjunto de acciones (o un subprograma) que se detalla luego por separado empleando otro diagrama de actividades. Se representa como una acción con un icono de una acción llamando a otra en la parte inferior derecha:

Las notas se emplean para comentar y documentar los diagramas de actividades. Se representan como una hoja con una esquina doblada y asociada, mediante una línea discontinua, a cualquiera de los elementos de un diagrama de actividades:



Al elaborar un diagrama de actividades se debe tener presente que es un lenguaje y como tal es necesario respetar su sintaxis (es decir los símbolos) pues cada uno de ellos tiene su propio significado y si se cambia, cambia también la lógica del algoritmo. Por ejemplo, para representar un flujo siempre se debe emplear la flecha continua abierta, no una flecha cerrada y rellena: ——►, una flecha cerrada no rellena: ——▷, o una flecha discontinua abierta: ····> pues cada una de ellas tienen un significado muy diferente (mensaje, herencia y dependencia respectivamente).

2.2 Diagrama de actividades de una secuencia

La *secuencia* es la estructura básica para la elaboración de cualquier programa. Una secuencia es una serie de instrucciones que se ejecutan una a continuación de otra. El diagrama de actividades de una secuencia se ve aproximadamente como se muestra en la siguiente figura:

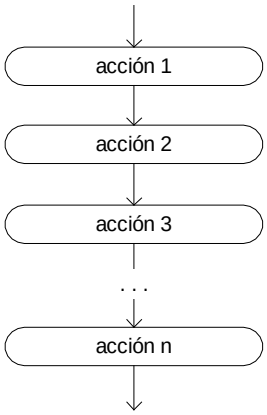


Figura 2.1. Diagrama de actividades de una secuencia

En este diagrama las acciones pueden ser instrucciones simples (como instrucciones de asignación) o instrucciones compuestas (como otra estructura estándar o inclusive otra secuencia). En consecuencia en *programación estructurada*, todo programa, subprograma o módulo es en última instancia una secuencia.

Debido a lo anterior, una secuencia puede ser representada también, de manera abreviada, como una actividad.

2.3 Codificación de una secuencia en hecl

En hecl, al igual que en C, las secuencias se encierran entre llaves. Las instrucciones se separan con saltos de línea, o si están en la misma línea con puntos y comas:

```
{
  instrucción 1
  instrucción 2
  ...
  instrucción n
}
```

O

```
{
  instrucción 1; instrucción 2; instrucción 3
  instrucción 4; instrucción 5
  ...
  instrucción n-1; instrucción n
}
```

2.4 Módulos

Para aplicar el primer principio de la programación estructurada debemos dividir el problema en problemas más pequeños que puedan ser resueltos independientemente. Como recordarán, estos problemas independientes se conocen como módulos y en hecl, los módulos se implementan en forma de procedimientos empleando la palabra reservada **“proc”**.

La estructura de un procedimiento en hecl es la siguiente:

proc *nombre_del_procedimiento* {*parámetros*} {

  instrucciones del procedimiento

}

El nombre del procedimiento, como en la mayoría de los lenguajes, puede estar conformado por letras y números (sin espacios), sin embargo, a diferencia de muchos otros lenguajes, en hecl, es posible declarar un procedimiento con el mismo nombre de uno ya existente, por lo que se debe tener cuidado para no sobre escribir alguna de las funciones de la librería de hecl.

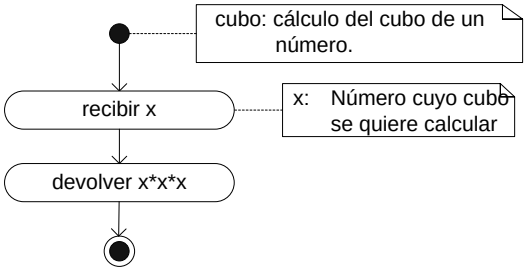
Los parámetros son los datos que recibe el módulo. Como en otros lenguajes, los parámetros son los nombres temporales que se dan a los datos que recibe el módulo. En hecl, los parámetros siempre se pasan por valor, es decir que normalmente no es posible modificar los valores de las variables originales.

Si el procedimiento consta de un solo parámetro no es necesario escribir las llaves. Por el contrario, si el procedimiento no tiene ningún parámetro las llaves deben ser escritas sin nada en su interior.

Para salir de un procedimiento y devolver un resultado, en hecl se emplea la instrucción “return” seguida del valor a devolver, sin embargo, si el valor a devolver es el último valor calculado, no es necesario emplear la palabra “return”.

Por ejemplo si queremos crear un módulo que calcule el cubo de un número cualquiera, la lógica a seguir es la siguiente: a) recibir el valor cuyo cubo se quiere calcular; b) multiplicar 3 veces el valor recibido; c) Devolver el resultado de la multiplicación. Observe que la resolución de este problema corresponde a la de un módulo: es independiente pues para resolver este problema no se requiere conocer de donde viene el número cuyo cubo se va a calcular y tampoco necesitamos saber donde se utilizará el resultado calculado.

El algoritmo en forma de diagrama de actividades es el siguiente.



Y el código respectivo sería el siguiente:

```
hecl> proc cubo x {* $x $x $x}
```

Este procedimiento tiene el nombre “cubo”, como sólo recibe un parámetro “x”, no se han empleado llaves para encerrar el mismo y dado que la última y única operación del procedimiento es el resultado buscado, no se requiere emplear el comando “return”.

Observe también que en hecl, para acceder al valor almacenado en una variable, se escribe el símbolo “$” delante del nombre de la variable.

Haciendo correr el programa con 3 y 5.4 se obtiene:

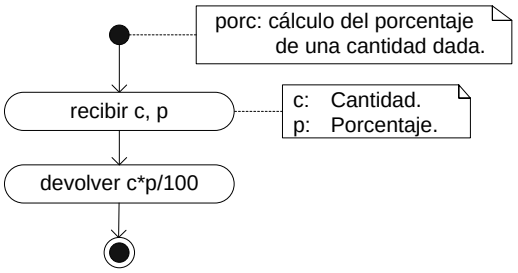
```
hecl> cubo 3
27
hecl> cubo 5.4
157.46400000000003
```

Que son los resultados correctos, con excepción del 3 después de los ceros en el resultado real, el cual se debe a un error de redondeo. Algo que ocurre normalmente en las operaciones de punto flotante (números reales).

Como otro ejemplo elaboremos un módulo que calcule un determinado porcentaje de una cantidad dada. Por ejemplo si la cantidad es 345.323 y se quiere calcular el 35% del mismo, la operación a efectuar sería la siguiente: 345.323*35/100 = 120.86305.

La lógica a seguir sería la siguiente: a) Recibir la cantidad y el porcentaje; b) Multiplicar el valor por el porcentaje y dividir entre 100; c) Devolver el valor calculado.

La lógica en forma de diagrama de actividades es la siguiente:



El código elaborado en hecl es:

```
hecl> proc porc { c p } { * $c [/ $p 100.] }
```

Haciendo correr el programa con 345.323, 35% y 12876, 63% se obtiene:

```
hecl> porc 345.323 35
120.86304999999999
hecl> porc 12876 63
8111.88
```

Es importante hacer notar que el punto (.) después del 100 en el procedimiento es imprescindible en hecl, pues como ya vimos anteriormente, si los números no tienen parte fraccionaria la división (/) sólo devuelve el cociente de la división entera.

Como ocurre en casi todos los lenguajes, los nombres de los parámetros no tienen mayor importancia, pues solo son nombres locales que desaparecen cuando el procedimiento concluye, sin embargo lo que es importante es el orden en que se mandan los mismos, así si en el anterior procedimiento siempre deberíamos mandar primero la cantidad y luego el porcentaje. Si bien en este caso en particular es posible invertir el orden (porque la operación que se realiza en el numerador es la multiplicación y la misma es conmutativa), se trata sólo de una casualidad que no debe servir de pauta para otros procedimientos.

2.5 Variables

Una variable es un espacio de memoria reservado al cual se le asigna un nombre (o identificador).

Para declara una variable en hecl se emplea la palabra reservada "set" de acuerdo al siguiente formato:

set *nombre_de_la_variable valor;*

Como se puede observar, en hecl, al ser un lenguaje interpretado, las variables se declaran y se inicializan con algún valor al mismo tiempo, además, dicho valor puede ser de cualquier tipo (numérico, carácter, etc.).

Por ejemplo, podemos hacer las siguientes declaraciones de variables:

```
hecl> set x 3.456
3.456
hecl> set y 6.753
6.753
hecl> set z Hola
Hola
hecl> set l { a b c d e f g }
 a b c d e f g
```

Para acceder a los valores almacenados en una variable se escribe el nombre de la variable precedido por el símbolo de dólar “$”:

```
hecl> puts $x
3.456
hecl> puts $y
6.753
hecl> puts $z
Hola
hecl> puts $l
 a b c d e f g
```

Con las variables podemos comprender con más claridad la función de las comillas en hecl, que como se dijo no identifican texto como suele ocurrir en otros lenguajes. Por ejemplo con las siguientes instrucciones obtenemos:

```
hecl> puts "$x $y $z $l"
3.456 6.753 Hola  a b c d e f g
hecl> puts "El resultado es: $x"
El resultado es: 3.456
hecl> puts "La lista es: $l y el reultado es: $y"
La lista es:  a b c d e f g  y el reultado es: 6.753
```

Además, en hecl, al no ser un lenguaje tipificado es posible reasignar a una variable un tipo de dato muy diferente al inicialmente asignado, así por ejemplo podemos reasignar a la variable “x” el valor “HOLA MUNDO”:

```
hecl> set x "HOLA MUNDO"
HOLA MUNDO
hecl> puts $x
HOLA MUNDO
```

Y como se ve no se produce ningún error, por lo que el control de tipos, en hecl, queda en manos del programador y es algo que se debe tomar muy en cuenta al momento de codificar los algoritmos.

2.6 Notepad++

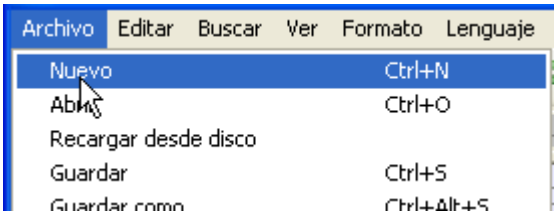
Como se sabe, el símbolo del sistema sólo permite hacer correcciones en una línea y dado que a partir de este tema elaboraremos procedimientos, es conveniente contar con una herramienta que facilite la escritura, mantenimiento y corrección del código elaborado.

En el celular se puede emplear el cuaderno de notas para esta labor, y en la computadora también podemos recurrir al mismo (Notepad), sin embargo en este ambiente existen algunas herramientas que pueden facilitar la tarea de crear y mantener programas. De entre ellas emplearemos en este curso Notepad++ (http://notepad-plus.sourceforge.net/). Un editor que puede reemplazar al editor de Windows y que soporta varios lenguajes, entre ellos tcl, que tiene algo así como un 90% de instrucciones compatibles con hecl, por lo que puede ser empleado para escribir programas en hecl. Es posible también añadir un nuevo lenguaje introduciendo la información necesaria con relación al mismo.

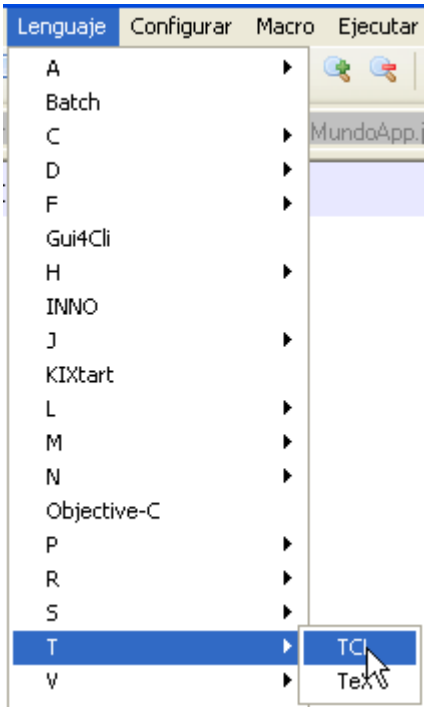
Básicamente este editor nos será de utilidad para guardar los programas elaborados (algo que no podemos hacer en el símbolo del sistema) y para corregir los errores cometidos en la elaboración de los programas.

Notepad++, cuenta con un instalador, por lo que la instalación del mismo en Windows, sigue la misma rutina que la instalación de cualquier otro software. También es posible bajar ejecutables del programa (que no requieren instalación), en ese caso, para ejecutar el programa se debe ingresar a la carpeta respectiva y hacer doble clic en el icono de Notepad++.

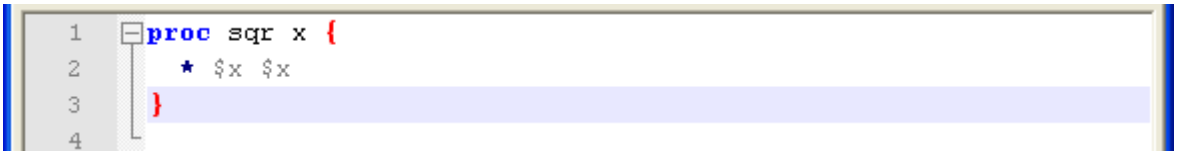
Una vez en Notepad++ lo primero que hacemos para comenzar a escribir programas es crear una nueva hoja de trabajo:



Luego elegimos el lenguaje de programación (TCL):



Y listo, ya podemos comenzar a escribir las instrucciones. Por ejemplo podemos escribir el código de un procedimiento que calcule el cuadrado de un número cualquiera:



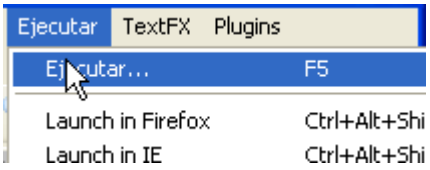
Por supuesto no es necesario escribir el programa en tres líneas, como se ha hecho, sin embargo, dado que en un editor podemos emplear más de una línea, es recomendable escribir los programas de la manera más clara posible. Eso sí, en hecl, para escribir un programa o instrucción en más de una línea, es necesario que exista algún símbolo en la línea o líneas previas que le indiquen al intérprete que la instrucción no ha sido aún completada. Es por eso que en la primera línea se deja la llave ({) abierta y recién se la cierra en la última línea.

Si en el anterior ejemplo se escribiera la llave en la segunda línea, hecl, trataría de ejecutar la primera línea como una instrucción independiente lo que daría lugar a un error.

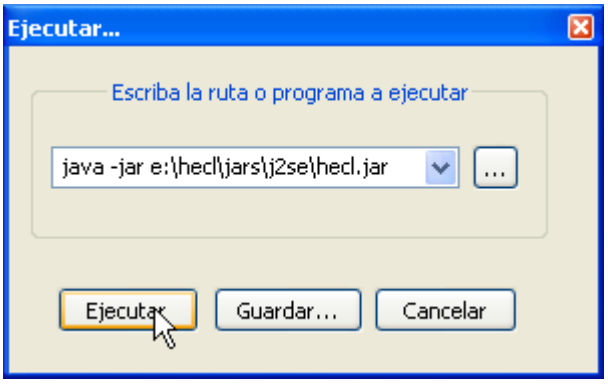
Observe que el editor resalta en otro color (en este caso azul) las palabras del lenguaje ("proc" en el ejemplo) y dado que TCL tiene en común un 90% de las instrucciones de hecl, este hecho ayuda a reducir el número de errores que se comete al escribir un programa. Observe también que cuando el cursor está sobre una llave, o corchete se resalta el par respectivo, lo que ayuda a evitar o corregir este tipo de error muy frecuente: no cerrar un corchete o una llave previamente abierta.

El editor diferencia también los operadores, números, variables, etc., facilitando de esa manera la elaboración y mantenimiento de los programas.

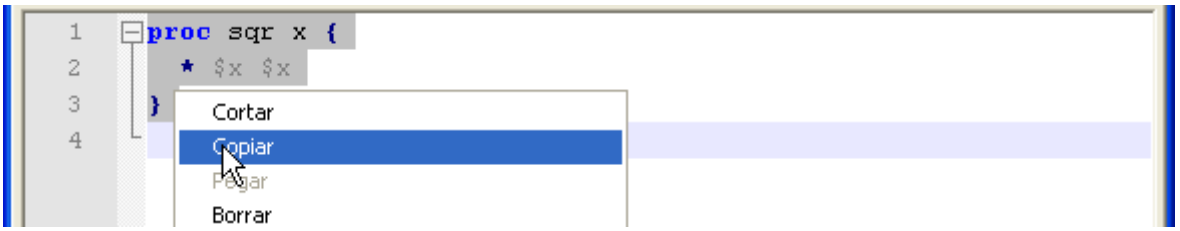
Una vez escrito el programa, el mismo debe ser copiado y ejecutado en hecl. Hecl puede ser ejecutado también desde el editor mediante la opción "Ejecutar" ("Run") del menú "Ejecutar":



Escribiendo la orden respectiva en la ventana que aparece:



Recuerde que "e:\hecl\jars\j2se\hecl.jar" es el camino y nombre del programa hecl. Una vez que hecl está corriendo, se copia el código escrito en Notpad++:



Y se copia el mismo en hecl (ctrl+V):

Entonces se puede emplear el procedimiento en la forma habitual, por ejemplo para calcular el cuadrado de 7.2:

A partir de este tema, procederemos de esa manera con los diferentes procedimientos que creemos, es decir escribiremos los procedimientos en Notepad++, los ejecutaremos en hecl y corregiremos y/o modificaremos los procedimientos en Notepad++.

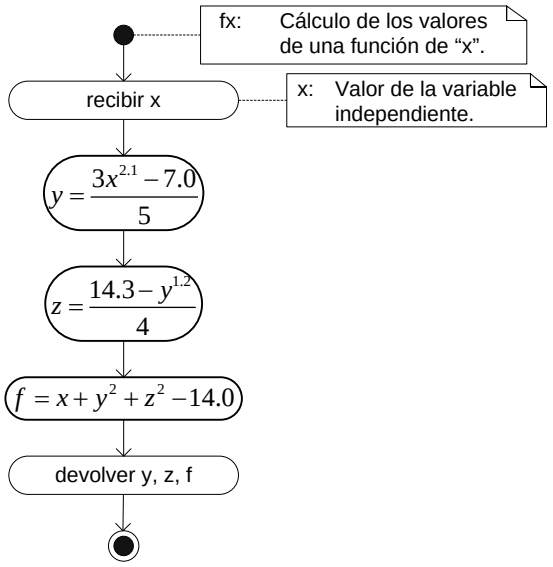
2.7 Ejemplos

1. Elabore un módulo que dado el valor de "x", calcule el valor de la siguiente función así como los valores de "y" y "z".

$$f(x) = x + y^2 + z^2 - 14.0 = 0$$
$$y = \frac{3x^{2.1} - 7.0}{5}$$
$$z = \frac{14.3 - y^{1.2}}{4}$$

En expresiones que tienen más de una ecuación, como esta, es importante determinar el orden en que deben llevarse a cabo las operaciones. Así en este caso no se puede calcular primero el valor de la función (f(x)), pues no se cuenta con los valores de "y" y "z", en consecuencia primero se deben calcular dichos valores y finalmente el valor de la función.

Por lo tanto la lógica para resolver este problema es la siguiente: a) Recibir el valor de "x". b) Con la segunda ecuación calcular el valor de "y"; c) Con la tercera ecuación calcular el valor de "z"; d) Calcular el valor de la función "f(x)"; e) Devolver los valores calculados. La lógica en forma de diagrama de actividades es la siguiente:



El código correspondiente es:

```
5  proc fx x {
6     set y [/ [- [* 3 [pow $x 2.1]] 7.0] 5]
7     set z [/ [- 14.3 [pow $y 1.2]] 4]
8     set f [+ $x [* $y $y] [* $z $z] -14.0]
9     list $y $z $f
10 }
```

Copiando este código en hecl, podemos probar el módulo con algunos valores:

```
hecl> fx 1.4
-0.18375763970384895 NaN NaN
hecl> fx 1.6
0.20991599022431942 3.536594392866494 0.15156462260658188
hecl> fx 2.1
1.4497828550453455 3.184605617262932 0.34358326428605324
hecl> fx 3.2
5.501860940941057 1.6405765868638549 22.161965350[illegible]8676
hecl> fx 4.5
12.722042992659004 -1.7144222693062927 155.28962162455738
```

Como se puede ver, en algunos casos la respuesta puede ser “NaN”, que es la forma en la cual hecl nos informa que el valor no ha podido ser calculado. Esto se puede deber a que existe una división entre cero, a que el resultado es infinito o a que el resultado es imaginario.

2. Elabore módulo que reciba los valores de “x_i”, “x” y “y” y devuelva el valor de la siguiente función.

$$f(x_i, x, y) = \frac{k_x}{k_y} + \frac{y - y_i}{x - x_i}$$

$$y_i = -0.0405 + 1.03785714286 x_i$$

$$k_x = \frac{k'_x}{(1-x)_{im}}$$

$$k_y = \frac{k'_y}{(1-y)_{im}}$$

$$(1-x)_{im} = \frac{(1-x)-(1-x_i)}{\ln\left(\frac{1-x}{1-x_i}\right)}$$

$$(1-y)_{im} = \frac{(1-y_i)-(1-y)}{\ln\left(\frac{1-y_i}{1-y}\right)}$$

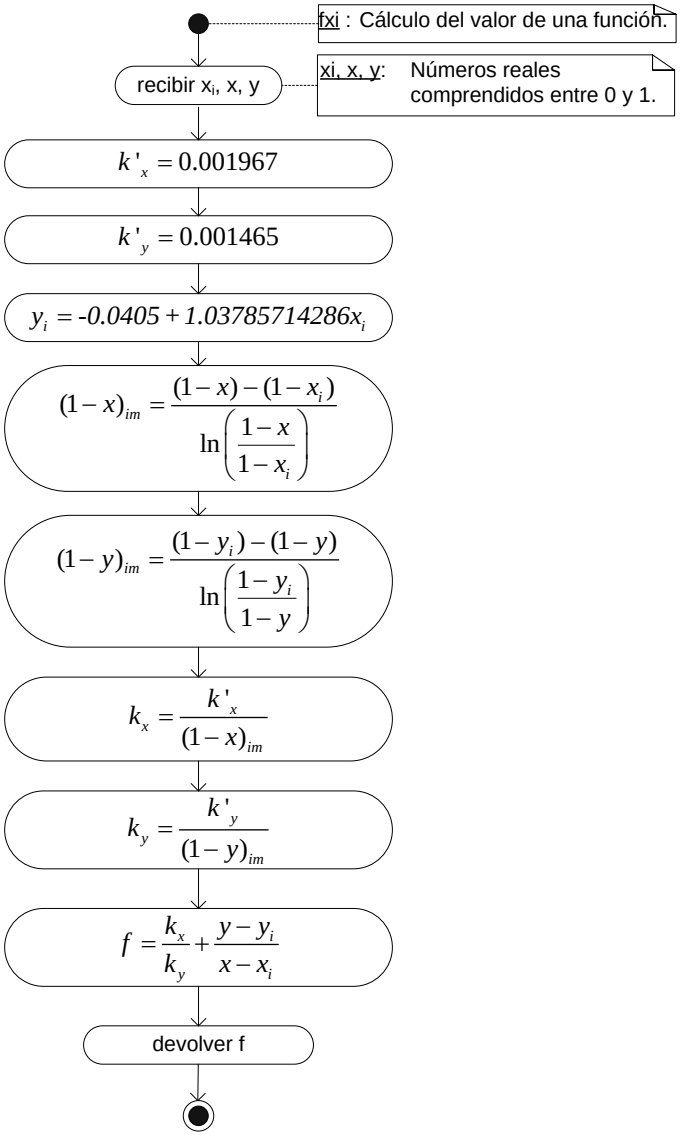
$$k'_x = 0.001967$$

$$k'_y = 0.001465$$

Al igual que el caso anterior es necesario determinar previamente el orden en que se llevaran a cabo los cálculos, no es posible por ejemplo calcular directamente el valor de la función pues no se conocen los valores de “k_x”, “k_y” ni “y_i” por lo que deben ser calculados previamente.

La lógica a seguir es la siguiente: a) Recibir los valores de “x_i”, “x” y “y”; b) Asignar a “k'_x” y a “k'_y” sus valores; c) Con la segunda ecuación calcular “y_i”; d) Con la quinta ecuación calcular $(1-x)_{im}$; e) Con la sexta ecuación calcular $(1-y)_{im}$; f) Con la tercera ecuación calcular “kx”; g) Con la cuarta ecuación calcular “ky”; h) Con la primera ecuación calcular el valor de la función; i) Devolver el valor de la función.

La lógica en forma de diagrama de actividades es la siguiente:



El código respectivo es el siguiente:

```
proc fxi { xi x y } {
  set kpx 0.001967
  set kpy 0.001465
  set yi [+ -0.0405 [* 1.03785714286 $xi]]
  set xim [/ [- [- 1 $x][- 1 $xi]][log [/ [- 1 $x][- 1 $xi]]]]
  set yim [/ [- [- 1 $yi][- 1 $y]][log [/ [- 1 $yi][- 1 $y]]]]
  set kx [/ $kpx $xim]
  set ky [/ $kpy $yim]
  set f [+ [/ $kx $ky][/ [- $y $yi][- $x $xi]]]
  return $f
}
```

Observe que en algunas de estas operaciones el orden no es importante, por ejemplo el valor de "k_y" puede ser calculado antes que el de "k_x" (porque el valor de "k_x" no se emplea en "k_y" o viceversa) mientras que, en

otras el orden es muy importante, por ejemplo antes de calcular $(1-y)_{im}$ es imprescindible calcular el valor de "y_i" (pues se emplea en dicho cálculo).

Se hace notar también que en la práctica no es necesario asignar el resultado de la expresión a la variable "f", pues esa variable sólo se emplea para devolver el resultado, por lo que dicho resultado podría ser devuelto directamente sin necesidad de ser almacenado previamente en una variable.

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> fxi 0.41 0.3 0.5
0.11392567069298054
hecl> fxi 0.51 0.4 0.47
1.4577562639666515
```

2.8 Ejercicios

1. Elabore un módulo que reciba el valor de "x" y devuelva los valores de las variables "y", "z", "w" así como el valor de la función.

$$f(x) = x + w + y - 39$$
$$x^{1/2} + z^2 = 35$$
$$y + e^{z/2} - \frac{8}{z} = 30$$
$$w + z^2 - 2z = 31$$

2. Elabore un módulo que reciba los valores de "p" y devuelva el valor de la siguiente función:

$$\text{f(p)=cos(kl)cosh(kl)+1}$$
$$a = \sqrt{EIg / (Ay)}$$
$$k = \sqrt{p / a}$$
$$l = 120$$
$$I = 170.6$$
$$E = 3x10^6$$
$$y = 0.066 \text{ lb/plg}^3$$
$$A = 32 \text{ plg}^2$$
$$g = 386 \text{ plg/s}^2$$

3. Elabore un módulo que reciba los valores de "x_1", "L_0", "x_0", "V_0" y "y_0" y devuelva el valor de la siguiente función:

$$f(x_1, L_o, x_o, V_o, y_o) = L_1 * x_1 + V_1 * y_1 - C_0$$
$$L_1 = \frac{L_c}{1 - x_1}$$
$$V_1 = M - L_1$$
$$y_1 = 1420 * x_1$$
$$L_c = L_0 * (1 - x_0)$$
$$M = L_0 + V_0$$
$$C_0 = L_0 * x_0 + V_0 * y_0$$

3. SELECCIÓN - 1

Como ya se mencionó en el tema anterior, sólo algunos problemas triviales pueden ser resueltos empleando únicamente la secuencia. Los problemas reales requieren al menos una estructura selectiva y frecuentemente una o más estructuras iterativas.

Continuando con las estructuras estándar, en este capítulo estudiaremos de las estructuras selectivas, el propósito del mismo es que al concluir el mismo estén capacitados para resolver problemas empleando las estructuras selectivas y siguiendo los principios de las programación estructurada.

El teorema de la programación estructurada señala que sólo se requiere una estructura selectiva (la estructura *IF-THEN-ELSE)* para resolver cualquier tipo de problema selectivo, sin embargo cuando la lógica involucra muchas condiciones consecutivas, el uso exclusivo de esta estructura da lugar a un algoritmo complejo, difíciles de entender y mantener, razón por la cual la mayoría de los lenguajes (incluido hecl) tienen alguna variante de esta estructura para esos casos.

Antes de comenzar el estudio de estas estructuras repasaremos brevemente algunos conceptos.

3.1. Expresiones lógicas

Una *expresión lógica* o *condición* es aquella que devuelve un valor lógico. Un valor lógico sólo puede ser o falso *(false)* o verdadero *(true)*.

Hecl tiene estos dos tipos de datos y numéricamente son equivalentes a 1 (true) y 0 (false):

```
hecl> puts [true]
1
hecl> puts [false]
0
```

Debemos recordar también que una expresión lógica se construye con operadores relacionales y/o operadores lógicos.

3.1.1. Operadores relacionales

Los operadores relacionales nos permiten comparar dos valores. En hecl podemos emplear los siguientes operadores relacionales:

>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
=	Igual a
!=	Diferente

Como ocurre con todos los operadores en hecl, se escriben primero los operadores y luego los valores a comparar, así las siguientes expresiones devuelven 1 (true) o 0 (false):

```
hecl> > 5 4
1
hecl> < 7 1
0
hecl> >= 6 5
1
```

```
hecl> <= 12 11
0
hecl> != 7 3
1
hecl> = 6 7
0
```

Recuerde que en el celular tiene escribir las anteriores instrucciones dentro de puts[…].

Si lo que se quiere comparar son expresiones matemáticas, es necesario evaluar primero las mismas (empleando corchetes), por ejemplo para realizar la siguiente comparación:

$$x\text{-}15+ln(z) > y*12$$

Donde x=2.3, y=4.5 y z=7.2, escribimos:

```
hecl> set x 2.3
2.3
hecl> set y 4.5
4.5
hecl> set z 7.2
7.2
hecl> > [+ $x -15 [log $z]][* $y 12]
0
```

Una vez más se recuerda que cuando se trabaja en el celular, para ver el resultado es necesario escribir la expresión dentro de la instrucción "puts".

3.1.2. Operadores lógicos

Los operadores lógicos, como su nombre sugiere, sólo trabajan con valores lógicos (falso o verdadero). Puesto que los operadores relacionales devuelven valores lógicos, pueden emplearse en combinación con los operadores lógicos para construir expresiones lógicas más ricas.

En hecl contamos con los siguientes operadores lógicos:

not	Negación lógica
and	Y lógico
Or	O lógico

El operador *not* cambia un valor lógico de falso *(false)* a verdadero *(true)* o viceversa.

```
hecl> not [true]
0
hecl> not [false]
1
```

O también:

```
hecl> not 1
0
hecl> not 0
1
```

El operador *and* devuelve verdadero si los dos valores que compara son verdaderos y falso en cualquier otro caso:

```
hecl> and [true] [true]
1
hecl> and [false] [false]
0
hecl> and [true] [false]
0
```

```
hecl> and [false] [true]
0
```

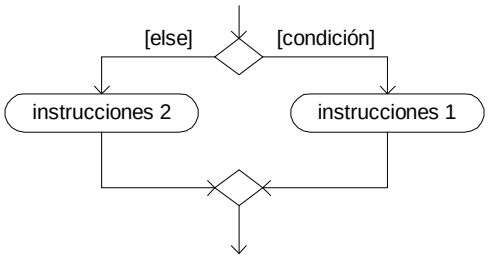
El operador *or*, devuelve falso si los dos valores que compara son falsos y verdadero en cualquier otro caso.

```
hecl> or [false] [false]
0
hecl> or [false] [true]
1
hecl> or [true] [false]
1
hecl> or [true] [true]
1
```

Una vez revisados estos conceptos, pasamos a estudiar la estructura selectiva if-then-else.

3.2. Estructura IF-THEN-ELSE

La estructura *IF-THEN-ELSE* tiene la lógica que se muestra en el siguiente diagrama de actividades:



Como se puede ver si la *condición* es verdadera se ejecutan las "instrucciones 1", caso contrario se ejecutan las "instrucciones 2". La forma de implementar esta estructura en *hecl* es la siguiente:

if *{condición} {instrucciones 1}* ***else*** *{instrucciones 2};*

Por ejemplo, la siguiente instrucción muestra el texto “Es Verdad”:

```
hecl> if {> 4 2} {puts "Es Verdad"} else {puts "Es Falso"}
Es Verdad
```

Mientras que la siguiente muestra el texto “Es Falso”:

```
hecl> if {< 4 2} {puts "Es Verdad"} else {puts "Es Falso"}
Es Falso
```

El caso contrario "else" de esta estructura es opcional, es decir puede no ser escrita, entonces si la condición es falsa, la instrucción "if" no hace nada y el control pasa directamente a la instrucción que se encuentre después del "if". Por ejemplo, la siguiente instrucción muestra directamente el resultado de sumar 7+10+11:

```
hecl> if {> 8 10} {puts "Es Mayor"}; + 7 10 11
28
```

Las "instrucciones" pueden ser también cualquier otra estructura estándar, incluida por supuesto la secuencia y la misma estructura *IF-THEN-ELSE.* En este último caso se dice que las estructuras están *anidadas.* Por ejemplo la siguiente instrucción tiene una estructura “if-then-else” anidada:

```
hecl> if {> 5 2} {if {!= 5 4} {puts "Then Interno"}}
Then Interno
```

A continuación se presentan algunos ejemplos donde se resuelven problemas empleando la estructura if-then-else.

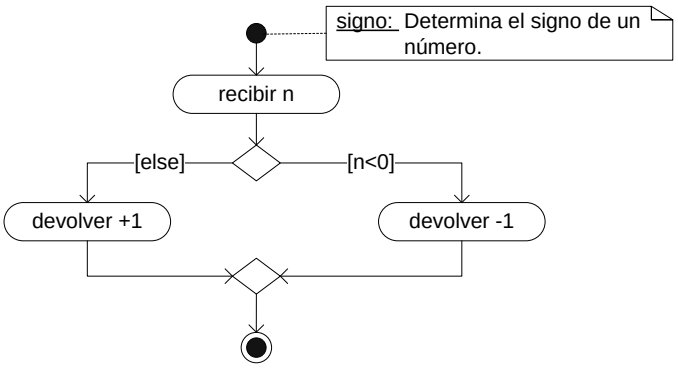
3.3. Ejemplos

1. Signo de un número

Como primer ejemplo, elaboraremos un módulo para determinar el signo de un número.

La lógica involucrada es muy simple, sólo se debe preguntar si el número es mayor a cero, en cuyo caso es positivo, caso contrario es negativo. Por supuesto también es posible hacer la pregunta inversa, es decir preguntar si el número es menor a cero, en cuyo caso es negativo y caso contrario es positivo. En cuanto al valor a devolver, para que sea de utilidad el mismo debe estar en formato numérico: "-1" cuando es negativo y "1" cuando es positivo.

El diagrama de actividades respectivo es el siguiente:



El código respectivo es el siguiente:

```
proc signo n {
   if {< $n 0} {return -1.0 } else {return 1.0}
}
```

Haciendo correr el módulo con 5.423 y -32.121 se obtiene:

```
hecl> signo 5.423
1.0
hecl> signo -32.121
-1.0
```

Que podemos comparar con los resultados devueltos por la función "signum" de hecl:

```
hecl> signum 5.423
1.0
hecl> signum -32.121
-1.0
```

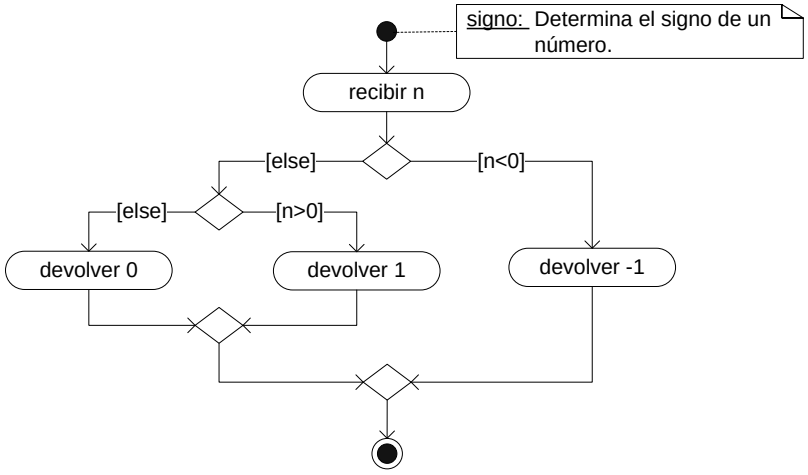
Sin embargo, si hacemos correr el módulo con "0" obtenemos:

```
hecl> signo 0
1.0
```

Mientras que con "signum" se obtiene

```
hecl> signum 0
0.0
```

Que es el resultado correcto pues “0” no es ni positivo ni negativo. En consecuencia es necesario corregir este aspecto en el módulo. Tomando en cuenta este caso el diagrama del módulo es:



El código respectivo es:

```
proc signo n {
  if {< $n 0} {
    return -1.0
  } else {
    if {> $n 0} {return 1.0} else {return 0.0}
  return 1.0}
}
```

Haciendo correr el módulo se obtienen los resultados correctos en todos los casos:

```
hecl> signo 5.423
1.0
hecl> signo -32.121
-1.0
hecl> signo 0
0.0
```

2. Año bisiesto

Como segundo ejemplo elaboraremos un módulo para determinar si un año dado es o no bisiesto. Por definición, un año es bisiesto si es divisible entre 100, excepto los años que terminan en dos ceros, los cuales deben ser divisibles entre 400.

Básicamente se trata de determinar si un número es divisible o no entre cuatro. Sin embargo, para resolver este problema, primero debemos considerar el caso especial, es decir cuando el año tiene dos ceros al final. De ser así, como tiene que ser divisible entre 400, es suficiente que le quitemos los dos ceros y con ello el caso especial se convierte en un caso normal, por ejemplo si el año es 1900, al quitarle los dos ceros se

convierte en 19 y con ello es suficiente comprobar si es divisible entre 4, pues el resultado de 19/4 es el mismo que de 1900/400:

Para determinar si un número es divisible o no entre otro, se calcula el residuo de la división (%), si dicho residuo es cero, el número es divisible, caso contrario no es divisible. Así por ejemplo 24 es divisible entre 8:

```
hecl> % 24 8
0
```

Mientras que no es divisible entre 7:

```
hecl> % 24 7
3
```

Igualmente para determinar si un número tiene o no dos ceros al final se calcula el residuo de su división entre 100 y si dicho residuo es cero, entonces el número tiene dos ceros al final, caso contrario no. Por ejemplo como 1800 tiene dos ceros al final, su residuo es cero:

```
hecl> % 1800 100
0
```

Mientras que el de 1920 no:

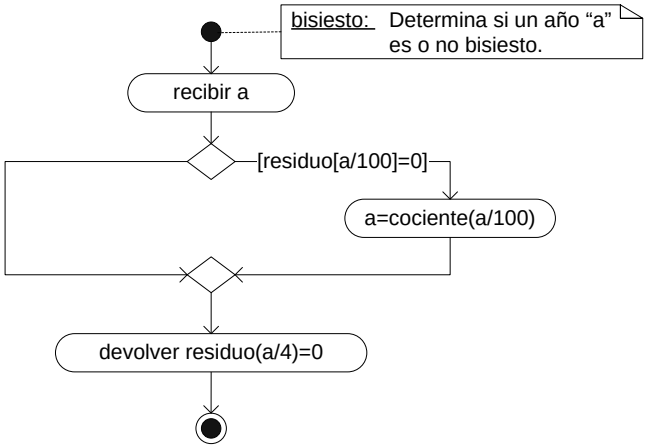
```
hecl> % 1920 100
20
```

Para quitar a un número dos (o el número de ceros que sea necesario), simplemente se calcula el cociente (/) de la división del número entre 100 (o 1000, 10000, según el número de ceros a quitar). Por ejemplo el cociente de 2400 entre 100 es:

```
hecl> / 2400 100
24
```

Es decir el número sin los dos ceros.

En cuanto a la respuesta, puesto que un año es o no es bisiesto (no hay años más o menos bisiestos), la respuesta debería ser de tipo lógica, es decir verdadera (1) si es bisiesto y falsa (0) caso contrario. El diagrama de actividades que toma en cuenta estas consideraciones es:



El código respectivo es:

```
proc bisiesto a {
   if {= [% $a 100] 0} {set a [/ $a 100]}
   = [% $a 4] 0
}
```

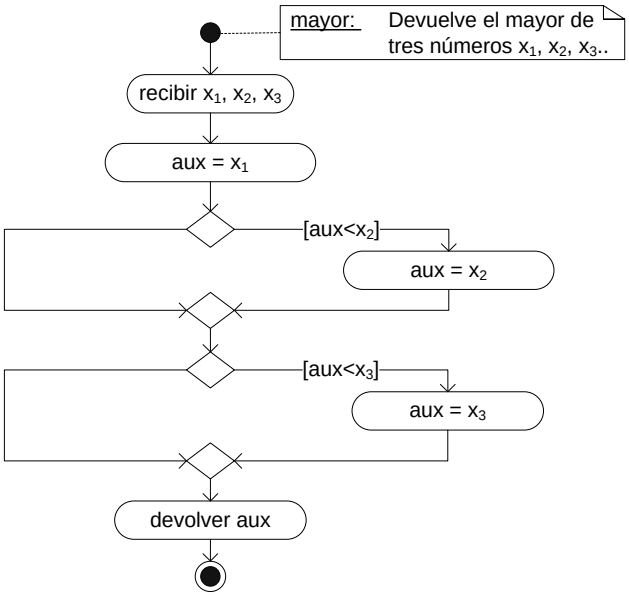
Haciendo correr el procedimiento con algunos valores se obtiene:

```
hecl> bisiesto 2000
1
hecl> bisiesto 1998
0
hecl> bisiesto 1900
0
hecl> bisiesto 2003
0
hecl> bisiesto 2004
1
hecl> bisiesto 2006
0
hecl> bisiesto 2008
1
hecl> bisiesto 2012
1
```

3. Mayor de tres números

Como tercer ejemplo elaboraremos un módulo que reciba tres números y devuelva el mayor de ellos.

Para resolver este problema, se puede emplear una variable auxiliar y en la misma se almacena el valor de la primera variable. Luego se compara dicha variable con las segunda y si es menor toma el valor de la segunda variable. Finalmente se compara con la tercera variable y si es menor hacer toma el valor de la tercera variable. De esa forma al concluir el proceso de comparaciones la variable auxiliar tendrá el mayor de los tres valores. El algoritmo en forma de diagrama de actividades es el siguiente:



El código respectivo es:

```
proc mayor { x1 x2 x3 } {
   set aux $x1
   if {< $aux $x2} {set aux $x2}
   if {< $aux $x3} {set aux $x3}
   return $aux
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

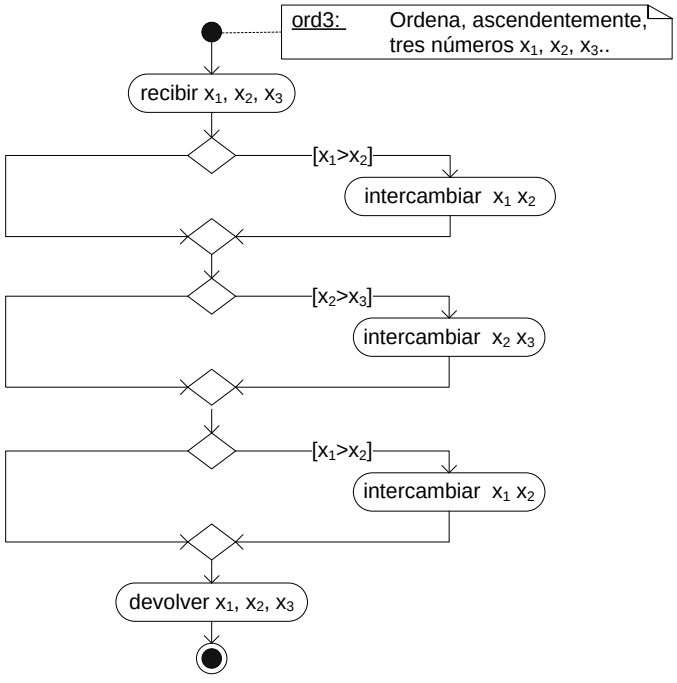
```
hecl> mayor 1 2 3
3
hecl> mayor 2 1 3
3
hecl> mayor 3 2 1
3
```

4. Ordenar dos números

Como cuarto ejemplo elaboraremos un módulo que reciba tres números y los ordene ascendentemente.

Para ordenar los tres números comparamos el primero con el segundo y si el primero es mayor que el segundo intercambiamos sus valores, luego comparamos el segundo con el tercero y si el segundo es mayor que el tercero intercambiamos sus valores, de esa manera nos aseguramos que en la tercera variable esté el mayor de los tres valores.

Ahora sólo falta verificar que las dos primeras variables tengan sus valores en orden ascendente, para ello volvemos a comparar la primera con la segunda variable y si la primera es mayor que la segunda intercambiamos sus valores. Con ello tenemos los tres números ordenados ascendentemente. El algoritmo en forma de diagrama de flujo es el siguiente:



El código respectivo es el siguiente:

```
proc ord3 { x1 x2 x3 } {
   if {> $x1 $x2} {set aux $x1; set x1 $x2; set x2 $aux}
   if {> $x2 $x3} {set aux $x2; set x2 $x3; set x3 $aux}
   if {> $x1 $x2} {set aux $x1; set x1 $x2; set x2 $aux}
   return [list $x1 $x2 $x3]
}
```

Haciendo correr el programa con los 6 posibles casos se obtiene:

```
hecl> ord3 3 2 1
1 2 3
hecl> ord3 3 1 2
1 2 3
hecl> ord3 2 1 3
1 2 3
hecl> ord3 2 3 1
1 2 3
hecl> ord3 1 3 2
1 2 3
hecl> ord3 1 2 3
1 2 3
```

Que como se puede observar devuelve los resultados correctos en todos los casos, por lo que podemos asumir que el programa ha sido correctamente evaluado.

Sin embargo, como se puede ver en el código, es necesario repetir el proceso de intercambio 3 veces, cuando ello sucede, se debe pensar en la posibilidad de crear un nuevo módulo para ese fin y dado que el intercambio de variables es un proceso relativamente frecuente es conveniente crear dicho módulo.

Para ello es necesario que aprendamos previamente como recibir y modificar los valores de variables en un procedimiento hecl.

En realidad en un procedimiento hecl las variables se reciben como cualquier otro valor, lo que cambia es el modo de tratar dichas variables dentro del procedimiento.

Básicamente lo que se tiene que hacer es evaluar expresiones dentro del procedimiento, indicándole el nivel en el que dichas expresiones deben ser evaluadas. En hecl, las expresiones pueden ser evaluadas con “eval”, por ejemplo para evaluar la expresión “+ 3 4 5” escribimos:

```
hecl> eval "+ 3 4 5"
12
```

Igualmente para evaluar la expresión + [* 1 2 3 4] [/ 8 4] escribimos:

```
hecl> eval {+ [* 1 2 3 4][/ 8 4]}
26
```

En otras palabras “eval” permite ejecutar instrucciones hecl que se encuentran dentro de listas o dentro de comillas, estas expresiones se conocen también con el nombre de “scripts”. Un comando que hace lo mismo que “eval”, pero que además permite especificar el nivel en el cual debe evaluarse la expresión es “upeval”. La instrucción “upeval” admite dos

parámetros: un número que permite especificar el nivel en el cual se evaluará la expresión y la expresión a evaluar.

El nivel se refiere a cuantos procedimientos se han llamado antes del procedimiento actual. Así, si el procedimiento ha sido llamado desde la línea de comando, el nivel en el que se encuentra el procedimiento es el 1, si por el contrario el procedimiento ha sido llamado desde otro procedimiento, se encuentra en el nivel 2, si ha sido llamado desde un procedimiento que a su vez ha sido llamado desde otro procedimiento, se encuentra en el nivel 3 y así sucesivamente.

Cuando se quiere evaluar la expresión en el nivel anterior al actual (es decir en el nivel del procedimiento que ha llamado al procedimiento actual) se debe escribir la instrucción “upeval” seguida del número -1, o dejar “upeval” sin número, porque por defecto “upeval” evalúa la expresión en el nivel anterior al actual. Si se quiere evaluar la expresión dos niveles previos al actual se debe escribir “upeval” seguido del número -2 y así sucesivamente. Si el número de nivel es 0, la expresión se evalúa a nivel de las variables globales, es decir en la raíz de hecl. Si el número es positivo, se evalúa en ese nivel específico, así si es 1 se evalúa en el nivel correspondiente a los procedimientos llamados desde la línea de comando, sin importar el nivel desde el cual se ejecuta el comando “upeval”.

Lo importante de la evaluación en uno u otro nivel es que la expresión se evalúa con las variables de dicho nivel y no con las actuales. Se debe recordar que en hecl, como ocurre en prácticamente todos los lenguajes de programación, las variables de un procedimiento son locales, es decir que sólo tienen validez mientras se está ejecutando el procedimiento.

A manera de ejemplo, analicemos los tres procedimientos siguientes:

```
proc a  {} {
   set var "Estoy en el nivel 1"
   b
}
proc b {} {
   c
}
proc c {} {
   upeval -2 {puts $var}
}
```

Llamando al procedimiento “a” se obtiene:

```
hecl> a
Estoy en el nivel 1
```

En el procedimiento “a” se crea la variable “var”, luego el procedimiento “a” llama al procedimiento “b”, que a su vez llama al procedimiento “c”. En este último procedimiento se evalúa la expresión “puts $var”, pero “var” no existe en este procedimiento, por lo que la instrucción generaría un error si no se ejecuta con “upeval -2”, que evalúa la expresión dos niveles antes, es decir el nivel correspondiente al procedimiento “a”, donde si existe la variable “var”.

Como otro ejemplo analicemos el siguiente procedimiento, que hace lo mismo que incr, es decir incrementa o disminuye le valor de la variable que se le manda en el número de dígitos especificado.

```
proc incre { x n} {
   set a [upeval [list set $x]]
   set a [+ $a $n]
   upeval [list set $x $a]
   return {}
}
```

Creando una variable (r) y haciendo correr el procedimiento con algunos valores de prueba se obtiene:

```
hecl> set r 1
1
hecl> incre r 2
hecl> set r
3
hecl> incre r 3
hecl> set r
6
hecl> incre r -4
hecl> set r
2
```

Donde se ha empleado “set” en lugar de “puts” para ver el valor de la variable después de llamar al procedimiento. En el procedimiento “incre” se recibe “x” (la variable) y el valor a incrementar (n). Entonces se recupera el valor de la variable “x” con “upeval [list set $x]” y se asigna dicho valor a la variable local “a”.

La expresión “upeval [list set $x]” opera de la siguiente manera: primero crea una lista con la instrucción “list” y los parámetros “set” y “$x”. Para la llamada con r resulta en: “{set r}” y que con otras variables tendrá el nombre de la variable respectiva. Como se sabe (y como se ha visto en el ejemplo) “set” y el nombre de la variable recupera el valor de la misma, y como esta orden es ejecutada en el nivel del procedimiento que hace la llamada, recupera el valor de la variable que se encuentra en dicho nivel (o de la línea de comando si es desde donde se hizo la llamada, como en el ejemplo).

Luego se le suma a la variable “a” el número “n” y dicho resultado es asignado de nuevo a la variable recibida en “x” mediante la instrucción “upeval [list set $x $a]”, que opera de manera similar al caso anterior, es decir primero forma la lista, que para el primer ejemplo resulta en {set r 3} y dicha expresión es evaluada en el nivel anterior, lo que resulta en la asignación del número 3 a la variable “r” del nivel anterior. Finalmente el programa devuelve una lista vacía “return {}” que en realidad es la forma empleada en hecl para no devolver nada.

Empleando este comando estamos en condiciones de elaborar un procedimiento para intercambiar el valor de dos variables, el razonamiento a seguir es bastante sencillo: a) recibir las dos variables a intercambiar; b) asignar el valor de la primera variable a una variable auxiliar local; c) asignar el valor de la segunda variable a la primera y d) asignar el valor de la variable auxiliar a la primera variable. Teniendo el cuidado de emplear “upeval” para evaluar las expresiones que involucran a las variables externas. El código respectivo es el siguiente:

```
proc inter { x y } {
  set aux [upeval [list set $x]]
  upeval [list set $x [upeval [list set $y]]]
  upeval [list set $y $aux]
  return {}
}
```

Asignando a la variable “x1” el valor 10, a “x2” el valor 20 y llamando al procedimiento se obtiene:

```
hecl> set x1 10
10
hecl> set x2 20
20
hecl> inter x1 x2
hecl> set x1
20
hecl> set x2
10
```

Ahora podemos volver a reescribir el procedimiento que ordena los tres números empleando el procedimiento “inter” para realizar los intercambios de variable:

```
proc ord3 { x1 x2 x3 } {
  if {> $x1 $x2} {inter x1 x2}
  if {> $x2 $x3} {inter x2 x3}
  if {> $x1 $x2} {inter x1 x2}
  return [list $x1 $x2 $x3]
}
```

Con el que se obtienen los mismos resultados que con la versión anterior.

5. Ecuación cuadrática

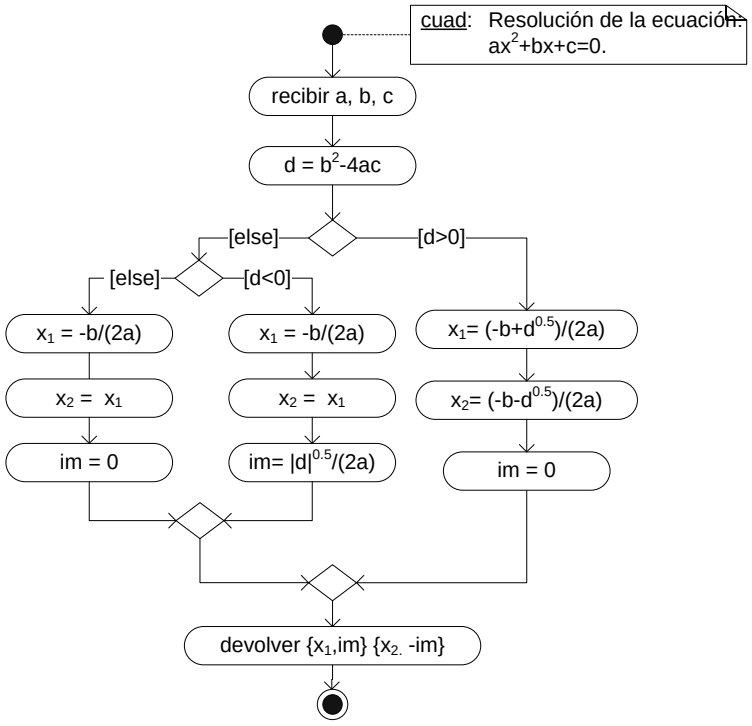
Como quinto ejemplo elaboraremos un módulo que resuelva una ecuación cuadrática, devolviendo tanto las soluciones reales como las imaginarias.

Como se sabe, la solución de la ecuación cudrática: $ax^2+bx+c=0$, se encuentra con la expresión:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Dependiendo del valor del discriminante: "*b2-4ac*", se pueden presentar uno de los siguientes casos: a) si el discriminantes es positivo, la raíz cuadrada tiene un valor real y en consecuencia se tienen dos soluciones reales y distintas (una con el signo + y otra con el signo -); b) si el discriminante cero la raíz cuadrada no tiene valor y en consecuencia las dos solucione son reales e iguales a –b/2a; c) si el discriminante es negativo la raíz cuadrada es imaginaria (pues es negativa) por lo tanto las soluciones son complejas (tienen parte imaginaria).

El algoritmo elaborado tomando en cuenta los tres casos descritos se presenta en el siguiente diagrama de actividades:



El código respectivo es el siguiente:

```
proc cuad {a b c} {
   set d [- [* $b $b][* 4 $a $c]]
   if {> $d 0} {
      set x1 [/ [- [* -1 $b][sqrt $d]][* 2 $a]]
      set x2 [/ [+ [* -1 $b][sqrt $d]][* 2 $a]]
      set im 0
     } else {
      if {< $d 0} {
        set x1 [/ [* -1 $b][* 2 $a]]
        set x2 $x1
        set im [/ [sqrt [abs $d]][* 2 $a]]
        } else {
          set x1 [/ [* -1 $b][* 2 $a]]
          set x2 $x1
          set im 0
        }
     }
    return [list [list $x1 $im][list $x2 [* -1 $im]]]
}
```

Haciendo correr el programa con datos para los tres casos posibles se obtiene:

```
hecl> cuad 1 -11 24
<3.0 0> <8.0 0>
hecl> cuad 1 -14 49
<7 0> <7 0>
hecl> cuad 1 2 3
<-1 1.4142135623730951> <-1 -1.4142135623730951>
```

3.4. Ejercicios

1. Elabore el diagrama de flujo y el código de un módulo que reciba un número entero y devuelva true, si el número es par y false en caso contrario.

2. Elabore el diagrama de flujo y el código de un módulo que reciba un número entero y devuelva true, si el número es impar y false en caso contrario.

3. Elabore el diagrama de flujo y el código de un módulo que reciba un número y determine si es par, impar o cero.

4. Elabore el diagrama de flujo y el código de un módulo que reciba un número y devuelva true si es un número entero y false en caso contrario.

5. Elabore el diagrama de flujo y el código de un módulo que reciba un número real y redondee el mismo al número inmediato superior si la parte fraccionaria es mayor o igual a 0.5 y al número inferior en caso contrario.

6. Elabore el diagrama de flujo y el código de un módulo que determine si tres lados dados conforman o no un triángulo: tres lados conforman un triángulo si la suma de cualesquier dos lados es siempre mayor al tercero.

7. Elabore el diagrama de flujo y el código de un módulo que reciba tres números y devuelva el menor de ellos.

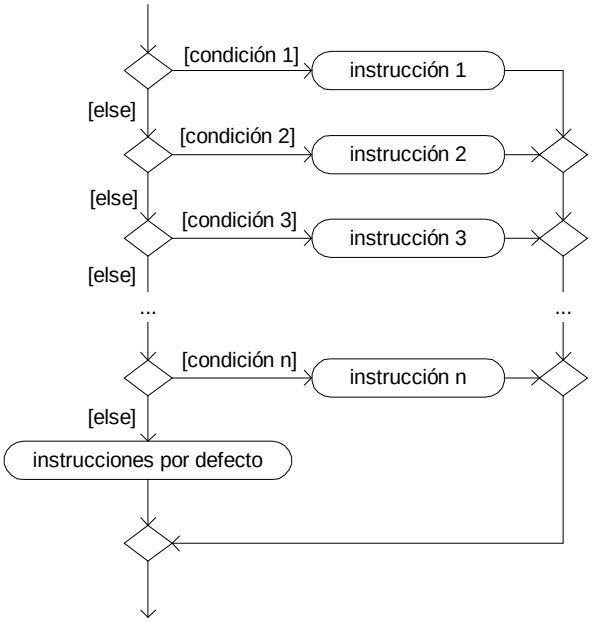
8. Elabore el diagrama de flujo y el código de un módulo que ordene descendentemente tres números.

9. Elabore el diagrama de flujo y el código de un módulo que reciba dos variables, conteniendo dos números, y sume a la primera variable el valor de la segunda, si el valor de la primera variable es mayor que el de la segunda y sume el valor de la primera variable a la segunda en caso contrario.

10. Elabore el diagrama de flujo y el código de un módulo que reciba dos variables, conteniendo dos números, y devuelva en la primera variable el menor de los valores y en la segunda el mayor.

4. SELECCIÓN - 2

Estudiemos ahora la segunda estructura selectiva: la estructura if-then-elseif. Esta estructura resulta más sencilla y eficiente cuando la lógica involucra varias condiciones consecutivas. El diagrama de actividades de esta estructura es el siguiente.



En esta estructura se llevan a cabo una serie de preguntas consecutivas ejecutándose la instrucción correspondiente a la primera condición verdadera. Una vez que se ejecuta una de las instrucciones el control sale de la estructura y continúa con el resto del programa. Si ninguna de las condiciones es verdadera se ejecutan las instrucciones por defecto, o no se ejecuta nada si no existen pues son opcionales.

En “hecl” la estructura if-then-elseif tiene la siguiente sintaxis:

```
if {condición 1} {
  instrucciones 1
} elseif {condición 2} {
  instrucciones 2
} elseif {condición 3} {
  instrucciones 3
} else {instrucciones por defecto}
```

Entonces si la lógica que resuelve el problema tiene la estructura mostrada en la figura, debería emplearse esta estructura en lugar de if-then-else, pues permite obtener un código más claro y en consecuencia más fácil de mantener.

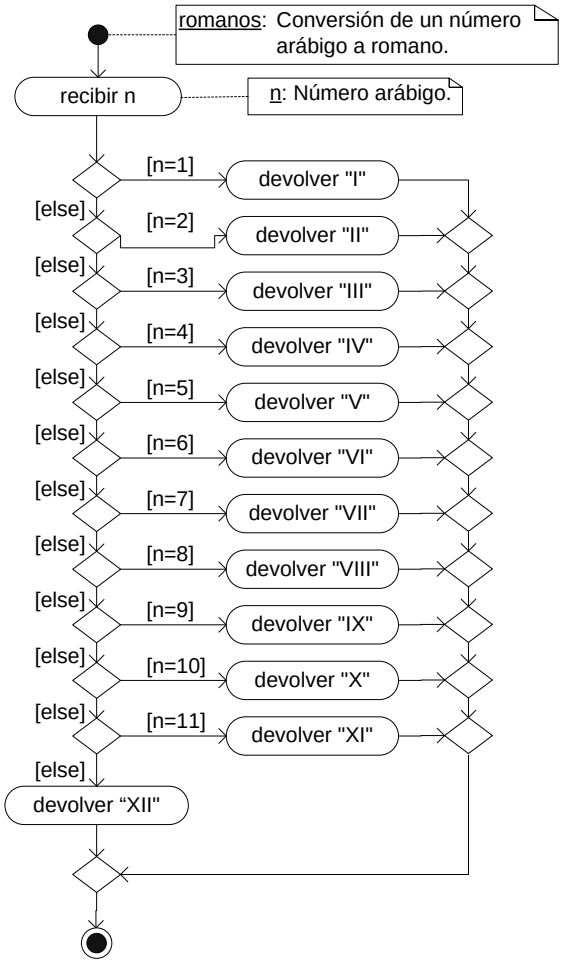
Un aspecto que es muy importante hacer notar es que para programar esta lógica es imprescindible que ninguno de los casos se superponga, es decir que no existan dos o más condiciones que sean verdaderas al mismo tiempo.

4.1. Ejemplos

1. Números Romanos

Como primer ejemplo, elaboraremos un módulo que reciba un número entero comprendido entre 1 y 12 y devuelva el número romano respectivo.

La lógica es muy simple: si el número es 1, se devuelve I, si es 2, II, si es 3, III y así sucesivamente. El diagrama de actividades que expresa esta lógica es el siguiente:xº



El código respectivo es el siguiente:

```
proc romanos n {
   if {!= [- $n [int $n]] 0} {throw "Numero real"}
   if {or [< $n 1][> $n 12]} {throw "Fuera de rango"}
   if {= $n 1} {
     return "I"
     } elseif {= $n 2} {
     return "II"
     } elseif {= $n 3} {
     return "III"
```

```
    } elseif {= $n 4} {
    return "IV"
    } elseif {= $n 5} {
    return "V"
    } elseif {= $n 6} {
    return "VI"
    } elseif {= $n 7} {
    return "VII"
    } elseif {= $n 8} {
    return "VIII"
    } elseif {= $n 9} {
    return "IX"
    } else {return "X"}
}
```

Como ya se dijo, hecl, no hace control de tipos, por lo que dicho control queda en manos del programador. Por eso en la primera parte del programa, antes de entrar a la lógica que resuelve el mismo, se verifica si el número tiene parte fraccionaria (pues tiene que ser un entero) y si tiene parte fraccionaria se genera un error con el comando “throw”. Luego se verifica también si el número es menor a 1 o mayor a 12, y de ser así también se genera un error, pues son valores que están fuera de los límites con los que opera el módulo.

Haciendo correr el programa con algunos valores de prueba se obtiene:

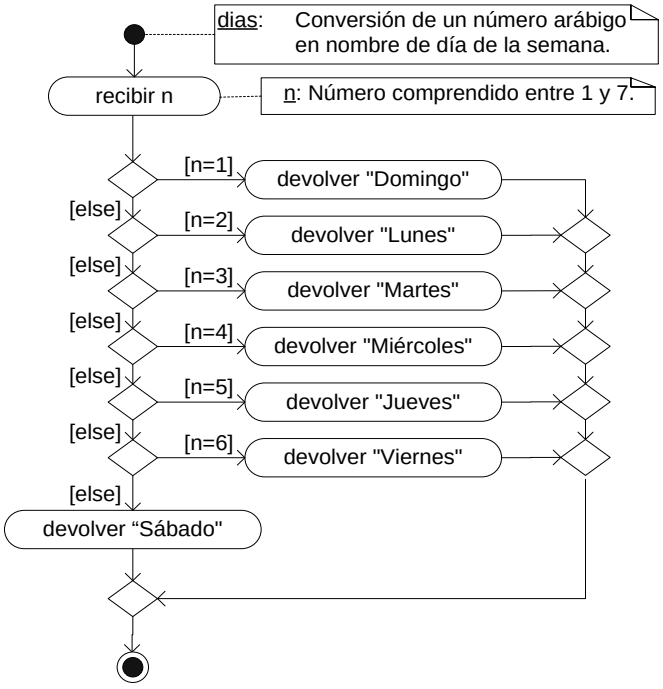
```
hecl> romanos 2
II
hecl> romanos 7
VII
hecl> romanos 9
IX
hecl> romanos 10
X
hecl> romanos 12
X
hecl> romanos 0
(ERROR (N·mero >12 o <1)) (throw 2) (if 4) (romanos 1)
        at org.hecl.Interp.run(Unknown Source)
N·mero >12 o <1
hecl> romanos 13
(ERROR (N·mero >12 o <1)) (throw 2) (if 4) (romanos 1)
N·mero >12 o <1
hecl> romanos 4.32
(ERROR (El n·mero no es entero)) (throw 2) (if 2) (romanos 1
)
        at org.hecl.Interp.run(Unknown Source)
El n·mero no es entero
```

Donde los mensajes de error han sido recortados para mostrar sólo las partes más importantes de los mismos.

2. Días de la semana

Como segundo ejemplo elaboraremos un módulo para convertir un número comprendido entre el 1 y 7 en el día de la semana respectivo.

La lógica que resuelve este problema es esencialmente la misma que la del anterior problema: si el número es 1 se devuelve “domingo”, si es 2 “lunes” y así sucesivamente, el diagrama de actividades respectivo es:



El código respectivo, haciendo las mismas consideraciones que en el ejemplo anterior, es:

```
proc dia n {
   if {!= [- $n [int $n]] 0} {throw "Número Real"}
   if {or [< $n 1][> $n 7]} {throw "Valor fuera de Rango"}
   if {= $n 1} {
     return "Domingo"
     } elseif {= $n 2} {
     return "Lunes"
     } elseif {= $n 3} {
     return "Martes"
     } elseif {= $n 4} {
     return "Miércoles"
     } elseif {= $n 5} {
     return "Jueves"
     } elseif {= $n 6} {
     return "Viernes"
     } else {
     return "Sábado"
     }
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> dia 1
Domingo
hecl> dia 4
Miúrcoles
hecl> dia 5
Jueves
```

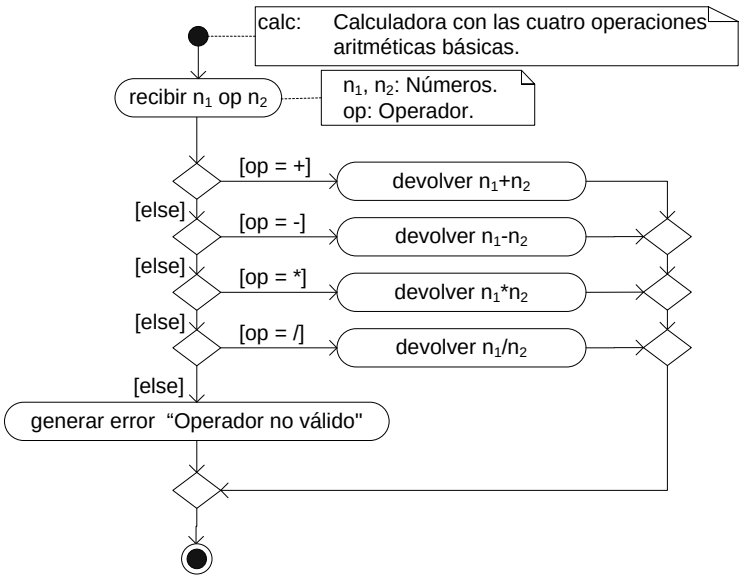
```
hecl> dia 7
SΩbado
hecl> dia 0
(ERROR (Valor fuera de Rango)) (throw 1) (if 3) (dia 1)
        at org.hecl.Interp.run(Unknown Source)
Valor fuera de Rango
hecl> dia 8
(ERROR (Valor fuera de Rango)) (throw 1) (if 3) (dia 1)
        at org.hecl.Interp.run(Unknown Source)
Valor fuera de Rango
hecl> dia 4.5
(ERROR (N·mero Real)) (throw 1) (if 2) (dia 1)
        at org.hecl.Interp.run(Unknown Source)
N·mero Real
```

Donde, al igual que en el anterior ejemplo, se han recortado los mensajes de error para mostrar sólo las partes más importantes.

3. Calculadora básica

Como tercer ejemplo elaboraremos un módulo que reciba tres parámetros: un número, un operador y otro número y con los mismos realice la operación especificada. El operador podrá ser de suma, resta, multiplicación o división.

El razonamiento para resolver este problema es el siguiente: se analiza el segundo parámetro (el operador) y si el mismo es “+”, se realiza la suma del primer y segundo parámetro, si es “-“ se realiza la resta y así sucesivamente. El algoritmo en forma de diagrama de actividades es el siguiente:



Siendo el código respectivo:

```
proc calc {n1 op n2} {
   if {catch {+ $n1 $n2}} {throw "Número no válido"}
   if {eq $op +} {
     return [+ $n1 $n2]
     } elseif {eq $op -} {
     return [- $n1 $n2]
```

```
    } elseif {eq $op *} {
    return [* $n1 $n2]
    } elseif {eq $op /} {
    return [/ $n1 $n2]
    } else {throw "Operador no válido"}
}
```

Donde, como se puede ver, se comprueba adicionalmente si los parámetros "n1" y "n2" son números, debido a que, como ya se dijo, hecl no hace control de tipos. Para ello en este caso hemos empleado la instrucción "catch", la cual permite "atrapar" un error, devolviendo "1" (verdadero) si se produce un error y "0" (falso) en caso contrario. Como está en una instrucción "if", si se producen un error "catch" devuelve 1 (verdadero) y en consecuencia se ejecuta se genera el error "Número de válido" (con throw), caso contrario devuelve 0 (falso) y en consecuencia pasa directamente a la siguiente instrucción (pues no existe la instrucción "else").

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> calc 4 + 5
9
hecl> calc 7 - 3
4
hecl> calc 8 * 9
72
hecl> calc 7 / 2.2
3.1818181818181817
hecl> calc 4 ^ 9
(ERROR (Operador no vßlido)) (throw 1) (if 3) (calc 1)
Operador no vßlido
hecl> calc a + b
(ERROR (N·mero no vßlido)) (throw 1) (if 2) (calc 1)
N·mero no vßlido
hecl>
```

Donde, como de costumbre, se han borrado las líneas adicionales que se muestran en la pantalla cuando se produce un error.

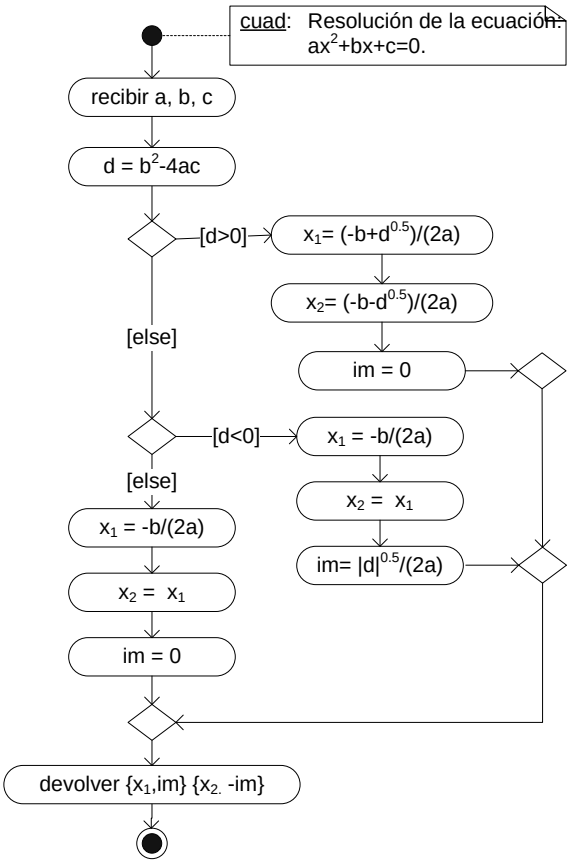
4. Ecuación cuadrática

Como cuarto ejemplo elaboraremos un módulo que resuelva una ecuación cuadrática, devolviendo tanto las soluciones reales como las imaginarias.

En este caso, el razonamiento es el mismo que en el tema anterior, sólo que en lugar de emplear la estructura if-then-else, para la implementación del programa, empleamos la estructura if-then-elseif y además hacemos un control de tipos. El algoritmo es el mismo que en el tema anterior, sólo que el diagrama de actividades ha sido reordenado para que tenga la apariencia correspondiente a la estructura if-then-elseif y es el que se presenta en la siguiente página.

El código respectivo es el siguiente:

```
proc cuad {a b c} {
    if {catch {+ $a $b $c}} {throw "Parámetro erróneo"}
    set d [- [* $b $b][* 4 $a $c]]
    if {> $d 0} {
```




```
      set x1 [/ [- [* -1 $b][sqrt $d]][* 2 $a]]
      set x2 [/ [+ [* -1 $b][sqrt $d]][* 2 $a]]
      set im 0
    } elseif {< $d 0} {
    set x1 [/ [* -1 $b][* 2 $a]]
    set x2 $x1
    set im [/ [sqrt [abs $d]][* 2 $a]]
    } else {
    set x1 [/ [* -1 $b][* 2 $a]]
    set x2 $x1
    set im 0
    }
   return [list [list $x1 $im][list $x2 [* -1 $im]]]
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> cuad 1 -11 30
{5.0 0} {6.0 0}
hecl> cuad 1 -10 25
{5 0} {5 0}
hecl> cuad 1 2 3
{-1 1.4142135623730951} {-1 -1.4142135623730951}
hecl> cuad a b c
{ERROR {Parßmetro err%neo}} {throw 1} {if 2} {cuad 1}
Parßmetro err%neo
hecl>
```

4.2. Ejercicios

1. Elabore el diagrama de flujo y el código de un módulo que reciba un número comprendido entre 1 y 4 y devuelva la estación del año respectiva, siendo el número 1 “primavera”.

2. Elabore el diagrama de flujo y el código de un módulo que reciba el nombre de un Departamento de Bolivia y devuelva la capital respectiva.

3. Elabore el diagrama de flujo y el código de un módulo que reciba el nombre de un mes y devuelva el número de días que tiene el mismo (asuma que febrero tiene 28 días).

4. Elabore el diagrama de flujo el código de un módulo que reciba el nombre de un mes y devuelva el número respectivo comenzando con 1 para el mes de enero.

5. Elabore el diagrama de flujo y el código de un módulo que reciba un número comprendido entre 1 y 100 y devuelva “reprobado” si el número es menor o igual a 50, “aprobado” si el número está comprendido entre 51 y 60, “regular” si el número está comprendido entre 61 y 70, “bien” si está comprendido entre 71 y 80, “muy bien” si está comprendido entre 81 y 90 y “excelente” si está comprendido entre 91 y 100.

6. Elabore el diagrama de flujo y el código de un módulo que reciba un número y devuelva “Sistemas” si el mismo es 35, “Informática” si es 26, “Civil” si es 32, “Química” si es 8, “Industrial” si es 37 y “Alimentos” si es 56.

5. ITERACIÓN - 1

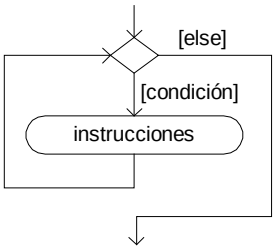
Para resolver problemas, no son suficientes las estructuras secuencial y selectiva. En muchos problemas es necesario repetir una serie de acciones un determinado número de veces o hasta que se cumpla una determinada condición, cuando esto ocurre, es decir cuando en la solución de un problema se repite una o más veces un determinado procedimiento, se dice que la solución es iterativa.

De acuerdo con el teorema de la programación estructurada (propuesto por Böhm y Jacopini), la única estructura necesaria para resolver problemas iterativos es la estructura *mientras (while)*.

En este tema estudiaremos dicha estructura y el objetito del tema es que al concluir el mismo estén en condiciones de resolver problemas iterativos aplicando la estructura while.

5.1. Lógica de la estructura While

El diagrama de actividades de la estructura *WHILE* es el siguiente:



En esta estructura, las instrucciones se repiten *mientras* la condición sea verdadera, terminan cuando la condición es falsa. Como se puede apreciar en el diagrama, si inicialmente la condición es falsa, las instrucciones no se repiten ni una sola vez.

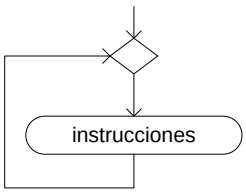
Como toda, estructura estándar, la estructura *WHILE* tiene un solo punto de entrada y uno de salida.

En hecl, esta estructura se codifica de la siguiente manera:

while *{condición} {instrucciones}*

Esta estructura se emplea cuando no se sabe el número de veces que debe repetirse el proceso iterativo y es especialmente indicada cuando existe la posibilidad de que dicho proceso no deba repetirse ni una sola vez.

Al emplear esta estructura se debe cuidar que la condición en algún momento, sea falsa, pues de lo contrario el ciclo se repetiría indefinidamente, dando lugar a lo que se conoce como un “ciclo infinito” y cuyo diagrama de actividades es el siguiente:



Para crear, intencionalmente, un ciclo infinito con la estructura *while*, simplemente se hace que la condición sea verdadera (true), es decir.

while *true {instrucciones}*

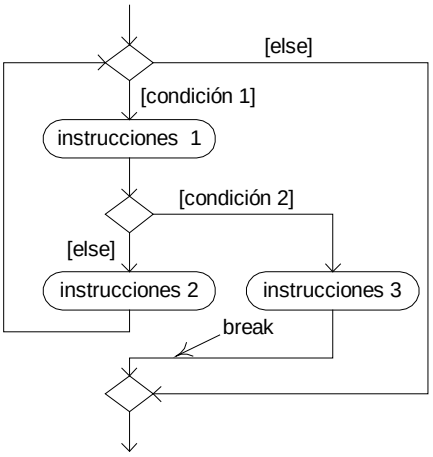
Aunque a primera vista puede parecer no tener sentido el crear un ciclo infinito para resolver un problema, en conjunción con las instrucciones que estudiaremos a continuación, este tipo de ciclo puede ayudar a resolver de forma más clara y eficiente algunos problemas.

5.2. Ruptura de un ciclo iterativo

En muchas ocasiones y sin importar la estructura iterativa que se esté empleando, resulta más eficiente y claro salir de un ciclo iterativo sin que se cumpla la condición del mismo. Para ello la mayoría de los lenguajes de programación cuenta con alguna instrucción que permite romper el ciclo. En el caso de hecl (y el de muchos otros lenguajes) dicha instrucción es *"break"*.

La instrucción *"break"* permite salir desde el interior de un ciclo iterativo y continuar la ejecución del programa con la instrucción que se encuentra a continuación de la estructura iterativa.

En el caso de la estructura *while*, la lógica de un ciclo iterativo con una ruptura del ciclo suele ser la que se muestra en el siguiente diagrama de actividades:



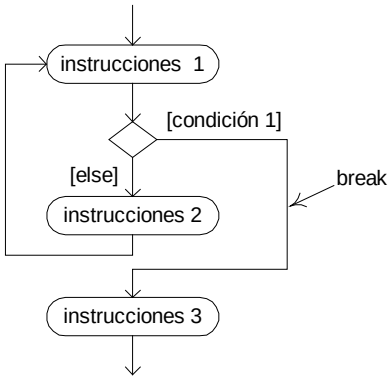
Donde sólo para ilustrar se ha señalado el flujo que corresponde a la ruptura del ciclo con *break*, no obstante, se recalca que esta instrucción no debe aparecer en un diagrama de actividades.

El código correspondiente a esta lógica tiene la siguiente forma:

```
while {condición 1} {
  instrucciones 1
  if {condición 2} {instrucciones 3; break}
  instrucciones 2
}
```

Con frecuencia la instrucción *break* se emplea para salir de un ciclo infinito, en cuyo caso, la lógica suele ser como se muestra en el diagrama de actividades de la siguiente página y cuyo código tiene la forma:

```
while true {
  instrucciones 1
  if {condición 1} break
  instrucciones 2
}
instrucciones 3
```

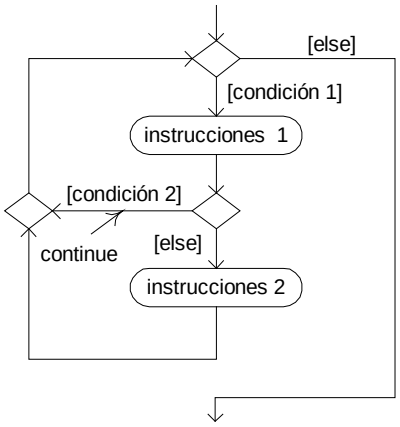


La instrucción "break" es la modificación que con mayor frecuencia se emplea en la práctica cuando se trabaja con ciclos iterativos.

5.3. Salto al siguiente ciclo iterativo

Cuando al interior de un ciclo se quiere saltar directamente a la siguiente iteración se puede emplear la instrucción *"continue"*. La instrucción *continue* es menos útil que la instrucción *break*, porque casi siempre se puede conseguir el mismo resultado con la estructura *if-then-else.* Sin embargo, cuando en la lógica existen muchos *if-then-else* anidados, la estruc*tura continue* puede resultar de utilidad.

Cuando se aplica esta modificación a la estructura *while*, la lógica suele ser la que se muestra en el siguiente diagrama de actividades:



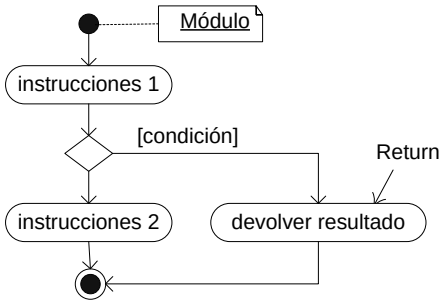
Que en hecl se codifica de la siguiente forma:

```
while {condición 1} {
  instrucciones 1
  if {condición 2} continue
  instrucciones 2
}
```

Como se puede observar en el diagrama, en lugar de la instrucción *continue,* se puede emplear fácilmente la instrucción *if-then-else* y eso es lo que normalmente sucede en la práctica.

5.4. Salida de un modulo

La mayoría de los lenguajes de programación, incluido hecl, cuentan con alguna instrucción que permite salir directamente del interior de un procedimiento o función. En el caso de hecl, dicha instrucción es *"return"*. Cuando hecl encuentra la instrucción *return*, termina la ejecución del módulo y devuelve como resultado del módulo el valor (o el resultado de la expresión) que se encuentra al lado de la instrucción (tal como ya se ha visto en los temas previos).



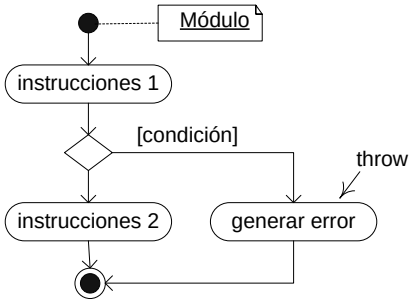
Normalmente, el código de un módulo que hace uso de esta lógica tiene la siguiente forma:

```
proc nombre {parámetros} {
  instrucciones 1
  if {condición} {return resultado}
  instrucciones 2
}
```

Adicionalmente, en hecl se puede salir de un módulo con la instrucción *exit,* no obstante *exit* termina de hecho la ejecución del entorno de *hecl,* no sólo del procedimiento, y en consecuencia no devuelve un resultado, sino un código que no puede ser utilizado dentro de hecl, sino desde otro lenguaje o entorno de programación.

5.5. Terminar un módulo generando un error

Ya hemos empleado esta forma de terminar un módulo en el tema anterior y como ya se vio se emplea sobre todo para controlar errores en tipos de datos o intervalos de valores válidos. Los módulos que emplean esta forma de concluir un módulo suelen tener una lógica similar al de la figura:



Y el código respectivo tiene aproximadamente la siguiente estructura:

```
proc nombre {parámetros} {
  instrucciones 1
  if {condición} {throw "error"}
```

instrucciones 2
}

El error generado con *throw* puede ser atrapado con "*catch":*

catch {instrucciones o procedimiento}

Catch devuelve 1 cuando al ejecutar las *instrucciones* (o procedimiento) se genera algún error, caso contrario devuelve 0. De esa forma, es posible averiguar si se ha producido o no un error al ejecutar un procedimiento y tomar acciones en función a ello.

5.6. Ejemplos

1. Raíz cuadrada de un número

Como primer ejemplo, elaboraremos un módulo para calcular la raíz cuadrada de un número real "n", empleando para ello la ecuación deducida con el método de Newton – Raphson:

$$x_2 = \frac{1}{2}\left(x_1 + \frac{n}{x_1} \right) \qquad (5.1)$$

El proceso básico que se sigue para calcular la raíz cuadrada del número "n" con esta ecuación es el siguiente:

a) Asumir un valor inicial para la solución: x_1.

b) Con x_1 y la ecuación de (5.1) calcular un nuevo valor de x: x_2.

c) Si x_1 y x_2 son aproximadamente iguales el proceso concluye, siendo la solución x_2.

d) Hacer que x_1 tome el valor de x_2: $x_1=x_2$ y repetir el proceso desde el paso (b).

El determinar cuando dos valores son aproximadamente iguales en una computadora, implica especificar en cuántos dígitos se quiere que dichos valores sean iguales. Para determinar si dos valores: x_1 y x_2, son iguales, en un determinado número de dígitos, se emplea la siguiente expresión lógica (condición):

$$\left| \frac{x_1}{x_2} - 1 \right| < 1x10^{-n}$$

Que devuelve *verdadero* si los valores comparados (x_1 y x_2) son iguales en los primeros *n* dígitos y *falso* en caso contrario. Así por ejemplo para determinar si las variables s_1 y s_2 son iguales en los primeros 6 dígitos la condición es:

$$\left| \frac{s_1}{s_2} - 1 \right| < 1x10^{-6}$$

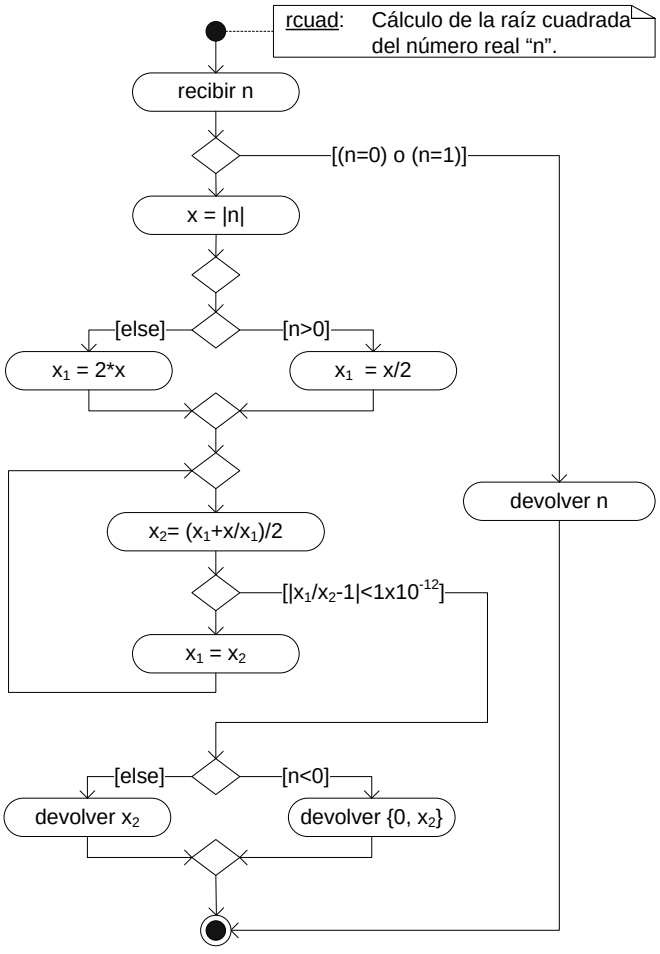
Por otra parte, en procesos iterativos con un número indeterminado de iteraciones, como en el presente, es importante comenzar con valores iniciales próximos a la solución. Para ello debemos notar que la raíz

cuadrada de números mayores a 1 son siempre menores al número, por ejemplo la raíz cuadrada de 12 es: 3.46410161514, mientras que la raíz cuadrada de número menores a 1 son siempre mayores al número, por ejemplo la raíz cuadrada de 0.3 es: 0.547722557505. En consecuencia y sin entrar en mayor refinamiento, se puede tomar como valor inicial la mitad del número (n/2) cuando el número es mayor a uno y el doble del número (2*n) en caso contrario.

Debemos tomar en cuenta también que la raíz cuadrada de números negativos es imaginaria, así por ejemplo, la raíz cuadrada de -4 es 0+2i. La ecuación de Newton – Raphson sólo permite calcular la raíz cuadrada de números positivos (sólo soluciones reales), por lo que en el caso de números negativos, se debe calcular el resultado empleando el valor positivo (el valor absoluto) y con el mismo armar la solución compleja: 0+(resultado_positivo)i.

Además, existen algunos casos en los cuales no es necesario calcular la raíz, así por ejemplo si el número es 0 o 1, no es necesario ejecutar ningún cálculo pues se sabe que la raíz cuadrada de 0 es 0 y de 1 es 1 (es decir la solución es el número cuya raíz se quiere calcular).

Tomando en cuenta las anteriores consideraciones, el diagrama de actividades para calcular la raíz cuadrada de un número real (con 12 dígitos de precisión), es el siguiente:



Como se puede ver la lógica no corresponde a la estructura *while* estándar, sino a un ciclo infinito con una ruptura *(break)*, que como se dijo es el caso que más se presenta en la práctica. El código respectivo es el siguiente:

```
proc rcuad n {
   if {or [= $n 0][= $n 1]} {return $n}
   set x [abs $n]
   if {> $n 0} {set x1 [/ 2. $n]} else {set x1 [* 2. $n]}
   while true {
     set x2 [/ [+ $x1 [/ $x $x1]] 2.]
     if {< [abs [- [/ $x1 $x2] 1]] 1e-12} break
     set x1 $x2
   }
   if {< $n 0} {return [list 0 $x2]} else {return $x2}
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> rcuad 0
0
hecl> rcuad 1
1
hecl> rcuad 2
1.4142135623730951
hecl> rcuad -2
0 -1.4142135623730951
hecl> rcuad 65.34
8.083316151184487
```

Resultados que pueden compararse con los obtenidos con la función “sqrt” de hecl:

```
hecl> sqrt 0
0.0
hecl> sqrt 1
1.0
hecl> sqrt 2
1.4142135623730951
hecl> sqrt -2
NaN
hecl> sqrt 65.34
8.083316151184487
```

Como se puede observar, la función “sqrt” de hecl genera un error (NaN) cuando el número es negativo. A pesar de que la precisión en el módulo “rcuad” ha sido fijada en solo 12 dígitos, los resultados son casi los mismos que con “sqrt”, por supuesto la precisión puede ser mejorada incrementando el número de dígitos, por ejemplo a 15 (1e-15).

2. Invertir los dígitos de un número entero

Como segundo ejemplo elaboraremos un módulo para invertir los dígitos de un número entero positivo o negativo. Así si el número es 123456, el número con los dígitos invertidos será: 654321.

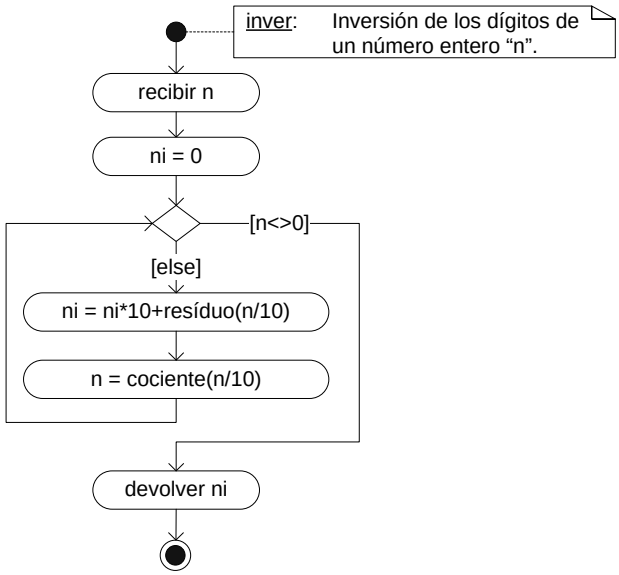
Para invertir los dígitos de un número entero primero debemos encontrar una forma de ir separando uno a uno los dígitos del número para irlos reacomodando en el sentido inverso. Para ello podemos emplear las operaciones de división entera, pues el cociente del número entre 10 nos de-

vuelve el número sin el último dígito, mientras que el residuo nos devuelve el último dígito.

Así por ejemplo el cociente de 123456 entre 10 es 12345, y el residuo es 6. Este último digito tiene que convertirse en el primero del número invertido. Si volvemos a calcular el cociente del número restante 12345, obtenemos 1234 y el residuo 5. Para que el 6 y el 5 (que son el primer y segundo valor del resultado) se vayan acomodando se puede multiplicar el primer dígito por 10 y sumar el segundo, es decir: 6*10+5 = 65.

Ahora si volvemos a repetir la operación con el número restante, el nuevo cociente es: 123 y el nuevo residuo 4. Ahora si multiplicamos 65*10 y le sumamos el último residuo obtenemos: 65*10+4=654. Prosiguiendo de esta manera hasta que el cociente de la división sea cero (es decir hasta que ya no queden dígitos en el número) podemos obtener el número invertido.

El diagrama de actividades que implemente la anterior lógica es el siguiente:



Como se puede observar, en este caso el ciclo iterativo corresponde a la estructura *while* estándar. El código respectivo es el siguiente:

```
proc invertir n {
  if {!= [- $n [int $n]] 0} {
    throw "El número a invertir debe ser entero"}
  set ni 0
  while {!= $n 0} {
    set ni [+ [* $ni 10][% $n 10]]
    set n [/ $n 10]
  }
  return $ni
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> invertir 12345
54321
hecl> invertir -65432543
-34523456
hecl> invertir 1
1
hecl> invertir 0
0
hecl> invertir 4543.434
(ERROR (El n·mero a invertir debe ser entero)) (throw 1) (if
 2) (invertir 1)
        at org.hecl.Interp.run(Unknown Source)
El n·mero a invertir debe ser entero
```

3. Cálculo del Fibonacci

Como tercer ejemplo elaboraremos un módulo para calcular el Fibonacci de un número entero. La ecuación que permite el cálculo del Fibonacci es: $F_n = F_{n-1}+F_{n-2}$; donde por definición $F_1 = F_2 = 1$.

El cálculo del Fibonacci comienza con el Fibonacci de 3:

$F_3 = F_2+F_1 = 1+1 = 2$

Luego, con este resultado se calcula el Fibonacci de 4:

$F_4 = F_3+F_2 = 2+1 = 3$

Con este resultado se calcula el Fibonacci de 5 y así sucesivamente hasta el número “n” cuyo Fibonacci se quiere calcular, por ejemplo, para “n=9” los cálculos necesarios son:

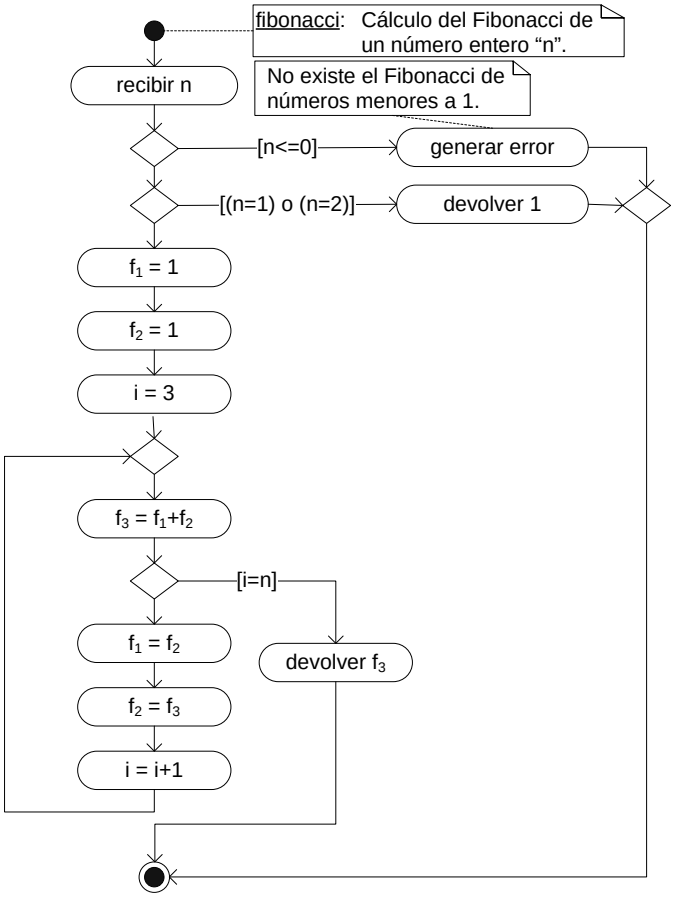
$F_5 = F_4+F_3 = 3+2 = 5$
$F_6 = F_5+F_4 = 5+3 = 8$
$F_7 = F_6+F_5 = 8+5 = 13$
$F_8 = F_7+F_6 = 13+8 = 21$
$F_9 = F_8+F_7 = 21+13 = 34$

Entonces, en el cálculo del Fibonacci, se sabe exactamente cuántas veces se debe repetir el procedimiento. Como veremos en el siguiente tema, cuando se conoce el número de veces que se debe repetir un procedimiento, la estructura iterativa más adecuada no es *while*, sin embargo, en este tema (debido a que estamos estudiando dicha estructura) resolveremos el problema recurriendo la misma.

Básicamente el algoritmo consiste en ir repitiendo los cálculos antes descritos, incrementando un contador en uno en cada repetición y realizando los cambios de variable respectivos, hasta que el contador es igual al número cuyo Fibonacci se quiere calcular.

El algoritmo que hace eso, en forma de diagrama de actividades, se presenta en la siguiente página y el código respectivo es:

```
proc fibonacci n {
    if {!= [- $n [int $n]] 0} {throw "El número debe ser entero"}
    if {<= $n 0} {throw "El número debe ser mayor a 0"}
    if {or [= $n 1][= $n 2]} {return 1}
    set f1 1
```




```
  set f2 1
  set i 3
  while true {
    set f3 [+ $f1 $f2]
    if {= $i $n} {return $f3}
    set f1 $f2
    set f2 $f3
    incr $i 1
  }
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> fibonacci 1
1
hecl> fibonacci 2
1
hecl> fibonacci 7
13
hecl> fibonacci 9
34
hecl> fibonacci 40
102334155
hecl> fibonacci -2
<ERROR <El n·mero debe ser mayor a 0>> <throw 1> <if 3> <fib
onacci 1>
```

```
El n·mero debe ser mayor a 0
hecl> fibonacci 7.6
<ERROR <El n·mero debe ser entero>> <throw 1> <if 2> <fibona
cci 1>
El n·mero debe ser entero
hecl>
```

4. Cálculo del seno hiperbólico

Como cuarto ejemplo elaboraremos un módulo para calcular el seno hiperbólico de un número real empleando la serie de Taylor:

$$senh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + ...\infty = \sum_{i=1,3,5,}^{\infty} \frac{x^i}{i!}$$

Que es una serie infinita, es decir que es imposible en la práctica calcular un resultado exacto con la misma (pues es imposible sumar hasta el infinito). Por ello, este tipo de series, se resuelve de manera iterativa y en cada iteración se suma un nuevo término hasta que el resultado de la sumatoria prácticamente no cambia, es decir hasta que los dos últimos valores calculados (el último y el penúltimo) son prácticamente iguales.

En cada iteración es necesario calcular el valor del término: ($x^i/i!$). Se podría pensar en calcular este nuevo valor elevando en cada iteración el número real a la potencia entera respectiva y dividiendo entre el factorial del número, es decir:

$$\frac{x^i}{i!} = \frac{x*x*x*x*...i\ veces}{1*2*3*4*...*i}$$

De esta manera se logra calcular el resultado correcto, pero la lógica es errónea pues se vuelven a calcular innecesariamente una y otra vez los mismos valores.

Para comprender el por qué este planteamiento es erróneo supongamos por ejemplo que se está calculado el término $x^{11}/11!$:

$$\frac{x^{11}}{11!} = \frac{x*x*x*x*x*x*x*x*x*x*x}{1*2*3*4*5*6*7*8*9*10*11}$$

Entonces en la siguiente iteración se debe calcular el término $x^{13}/13!$:

$$\frac{x^{13}}{13!} = \frac{x*x*x*x*x*x*x*x*x*x*x*x*x}{1*2*3*4*5*6*7*8*9*10*11*12*13}$$

Como se puede observar, para calcular $x^{13}/13!$ se vuelve a calcular el valor de $x^{11}/11!$:

$$\frac{x^{13}}{13!} = \left(\frac{x*x*x*x*x*x*x*x*x*x*x}{1*2*3*4*5*6*7*8*9*10*11}\right)\left(\frac{x*x}{12*13}\right) = \left(\frac{x^{11}}{11!}\right)\left(\frac{x^2}{12*13}\right)$$

Lo cual es ilógico porque ¿para qué se vuelve a calcular un valor que ya ha sido calculado previamente? Por supuesto este problema se repite desde los primeros términos, así, al calcular $x^{11}/11!$ se vuelve a calcular

$x^9/9!$; al calcular $x^9/9!$ se vuelve a calcular $x^7/7!$ y así sucesivamente. Proceso que como se ve es extremadamente ineficiente pues se vuelven a calcular una y otra vez los mismos valores, así por ejemplo cuando se calcula el término $x^{21}/21!$, se habrá calculado 10 veces el valor de $x^3/3!$, 9 veces el valor de $x^5/5!$ y así con los otros términos.

Lo lógico es utilizar los valores calculados previamente para calcular los nuevos, así por ejemplo, si ya se ha calculado el término $x^9/9!$, el nuevo término se calcula con:

$$\frac{x^{11}}{11!} = \frac{x^9}{9!} * \frac{x^2}{10*11}$$

Y una vez calculado $x^{11}/11!$, el nuevo término se calcula con:

$$\frac{x^{13}}{13!} = \frac{x^{11}}{11!} * \frac{x^2}{12*13}$$

En general entonces se puede calcular el nuevo término multiplicando el término anterior por x^2 y dividiendo entre 2 contadores que deberán incrementar de 2 en 2 en cada iteración:

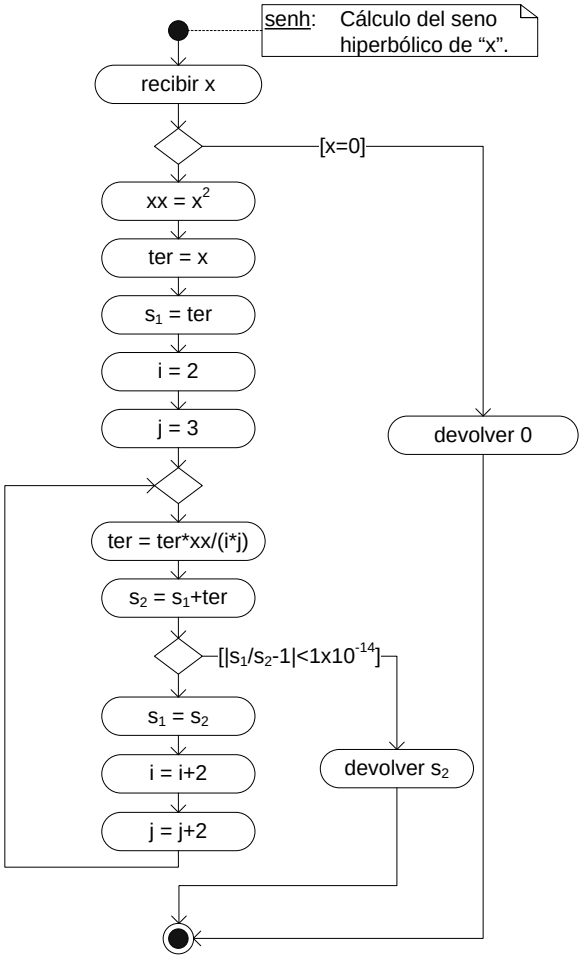
$$nuevo\ término = término\ anterior * \frac{x^2}{i * j}$$

Antes de comenzar el proceso iterativo se debe verificar que el número sea diferente de cero, por dos razones: a) Se sabe que el seno hiperbólico de cero es cero y b) Si "x" es cero, entonces "s_2" es cero, lo que daría lugar a un error por división entre cero al comparar las dos últimas sumatorias para determinar si son iguales en un determinado número de dígitos:

$$\left| \frac{s_1}{s_2} - 1 \right| < 1x10^{-n}$$

El algoritmo para calcular el seno hiperbólico se presenta en el diagrama de actividades de la siguiente página y el código respectivo es el siguiente:

```
proc senh x {
  if {= $x 0} {return 0}
  set xx [* $x $x]
  set ter $x
  set s1 $ter
  set i 2.
  set j 3.
  while true {
    set ter [* $ter [/ $xx [* $i $j]]]
    set s2 [+ $s1 $ter]
    if {< [abs [- [/ $s1 $s2] 1]] 1e-14} {return $s2}
    set s1 $s2
    set i [+ $i 2]
```




```
        set j [+ $j 2]
    }
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> senh 0
0
hecl> senh 4.5
45.003011151991785
hecl> senh 7.2
669.7150089043O5
hecl> senh 12.3
109847.99433379309
hecl> senh 34.7
5.874738318600534E14
hecl> sinh 0
0.0
hecl> sinh 4.5
45.003011151991785
hecl> sinh 7.2
```

Resultados que pueden ser corroborados con la función “sinh” de hecl:

```
hecl> sinh 0
0.0
hecl> sinh 4.5
45.003011151991785
hecl> sinh 7.2
669.7150089043048
hecl> sinh 12.3
109847.99433379307
hecl> sinh 34.7
5.874738318600539E14
```

Y como se puede ver, los resultados o son iguales o difieren a partir del décimo sexto decimal (pues la precisión ha sido fijada en 14 dígitos).

5. Cálculo del exponente

Como quinto ejemplo elaboraremos un módulo para calcular el exponente de un número real empleando la serie de Talor:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ...\infty, \quad -\infty < x < \infty$$

La lógica para resolver este problema es esencialmente la misma que la del anterior ejemplo y que prácticamente es la lógica que permite resolver cualquier serie: a) inicializar variables; b) calcular el nuevo término y el nuevo valor de la serie; c) comparar los dos últimos valores calculados y si son iguales en un determinado número de dígitos terminar el proceso, caso contrario hacer cambio de variables, incrementar contadores y repetir el proceso desde el paso (b).

No obstante, en este caso se debe tomar una medida adicional porque cuando "x" es negativo los términos con potencias impares son negativos y cuando en una serie aparecen restas los errores por redondeo incrementan considerablemente, a tal punto que el resultado calculado puede ser del todo erróneo, sin importar que la lógica sea correcta.

Por ello, se debe tratar de evitar el trabajar con sumas y restas cuando se resuelven series, o si no se puede evitar trabajar con número o muy grandes o muy pequeños. En este caso afortunadamente podemos evitar trabajar con números negativos: cuando "x" es negativo, se calcula el valor del exponente con el valor absoluto de "x", siendo la solución la inversa del valor calculado, pues se sabe que "e^{-x}" es igual a "$1/e^x$".

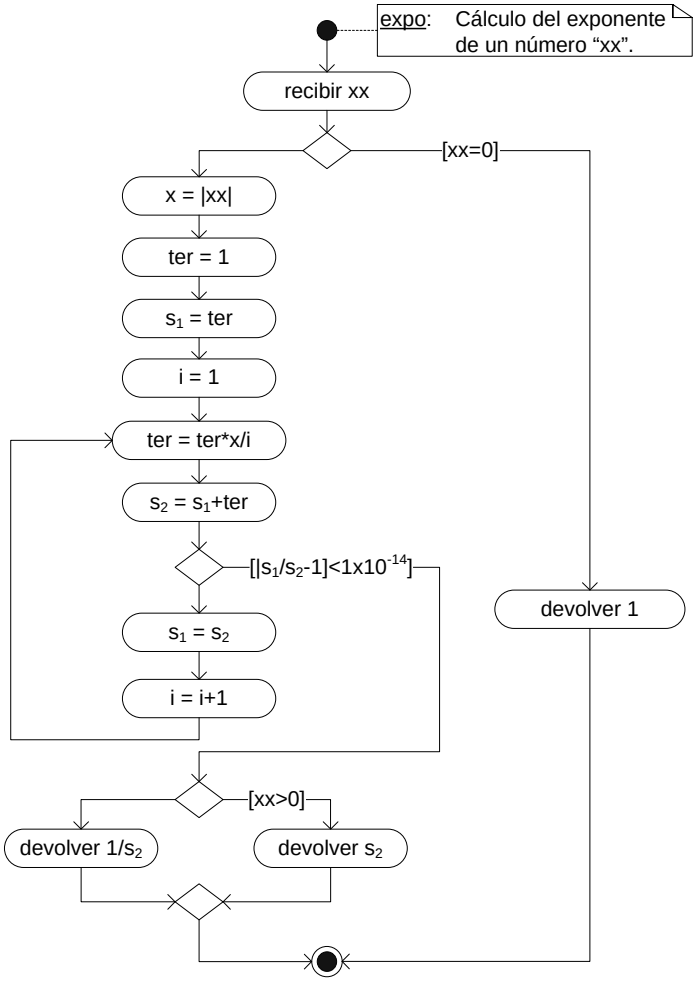
En este caso, los nuevos términos (para cada nueva iteración) se calculan multiplicando el término anterior por “x” y dividiendo entre un contador “i” que incrementa de uno en uno:

$$Nuevo\_Término = Término\_Anterior * \frac{x}{i}$$

Por ejemplo para generar el quinto término ($x^5/5!$) a partir del cuarto ($x^4/4!$), hacemos la operación:

$$\frac{x^5}{5!} = \frac{x^4}{4!} * \frac{x}{5}$$

El diagrama de actividades con la lógica que resuelve el problema, tomando en cuenta las anteriores consideraciones es:



El código respectivo es el siguiente:

```
proc expo xx {
   if {= $xx 0} {return 1}
   set x [abs $xx]
   set ter 1
   set s1 $ter
   set i 1.
   while true {
     set ter [/ [* $ter $x] $i]
     set s2 [+ $s1 $ter]
     if {< [abs [- [/ $s1 $s2] 1]] 1e-14} break
     set s1 $s2
     set i [+ $i 1]
   }
   if {> $xx 0} {return $s2} else {return [/ 1 $s2]}
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> expo 0
1
hecl> expo 2.1
8.16616991256765
hecl> expo 4.5
90.01713130052177
hecl> expo 12.45
255250.3226293341
hecl> expo -7.2
7.465858083766797E-4
hecl> expo -14.2
6.807981343976358E-7
```

Resultados que podemos corroborar con la función “exp” de hecl:

```
hecl> exp 0
1.0
hecl> exp 2.1
8.16616991256765
hecl> exp 4.5
90.01713130052181
hecl> exp -7.2
7.465858083766792E-4
hecl> exp 12.45
255250.32262933443
hecl> exp -14.2
6.807981343976342E-7
```

Como se puede observar, y al igual que el caso anterior, los resultados o son iguales o concuerdan en los 14 dígitos de precisión con los cuales se ha trabajado en el módulo.

5.7. Ejercicios

1. Elabore el diagrama de flujo y el código de un módulo que devuelva el número de dígitos de los que consta un número entero positivo o negativo.

2. Elabore el diagrama de flujo y el código de un módulo que devuelva, sin emplear la función “int”, la parte entera de un número real.

3. Elabore el diagrama de flujo y el código de un módulo que devuelva, sin emplear la función “int”, la parte fraccionaria de un número real.

4. Elabore el diagrama de flujo y el código de un módulo que calcule, con 9 dígitos de precisión, el seno de un ángulo con la serie: $sen(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...\infty, \quad -\infty < x < \infty$, donde el ángulo está en radianes. En el módulo se deben convertir los ángulos mayores a 2π a su equivalente comprendido entre 0 y 2π.

5. Elabore el diagrama de flujo y el código de un módulo que empleando la estructura "mientras" calcule el Chebyshev enésimo de un número real *x*, con la ecuación: $Ch_{n+1}(x) = 2xCh_n(x) - Ch_{n-1}(x);\ Ch_0(x) = 1;\ Ch_1(x) = x$.

6. Elabore el diagrama de flujo y el código de un módulo que calcule, con 15 dígitos de precisión, la raíz cúbica de un número real "n" con la fórmula de Newton: $x_2=(2x_1+n/x_1^2)/3$.

7. Elabore el diagrama de flujo el código de un módulo que calcule el arco tangente hiperbólico de un número real empleando la serie de Taylor: $\tanh^{-1}(x) = x+x^3/3+x^5/5+x^7/7+ ...\infty$; $|x|<1$.

8. Elabore el diagrama de flujo y el código de un módulo que calcule una de las soluciones de la ecuación cúbica: $ax^3+bx^2+cx+d=0$, con la ecuación de Newton: $x_2=(2ax_1^3+bx_1^2-d)/(3ax_1^2+2bx_1+c)$. El valor inicial asumido "x_1" debe ser uno de los parámetros. En el módulo se debe contar el número de veces que se repite el ciclo y si el mismo llega a 30, debe generar el error "Resultado no encontrado". Pruebe el módulo con las ecuaciones $x^3+2x^2+3x+4=0$ y $5x^3-9x^2-8x+5=0$.

9. Elabore el diagrama de flujo y el código de un módulo que calcule el Legendre enésimo de un número real "x", con la ecuación: $(n+1)Le_{n+1}(x)-(2n+1)xLe_n(x)+nLe_{n-1}(x)=0$ donde por definición: $Le_0(x)=1$, $Le_1(x)=x$.

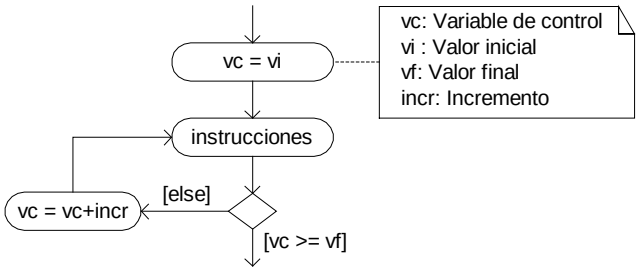
6. ITERACIÓN - 2

Si bien es cierto que cualquier problema iterativo puede ser resuelto empleando únicamente la estructura “while”, en ocasiones puede ser de utilidad emplear otras estructuras iterativas como es el caso de la estructura “for”.

El propósito de esta estructura es la de repetir una o más instrucciones un determinado número de veces, desde un valor inicial hasta un valor final. Por lo tanto esta estructura resulta más adecuada cuando se sabe el número de veces que debe repetirse un determinado proceso.

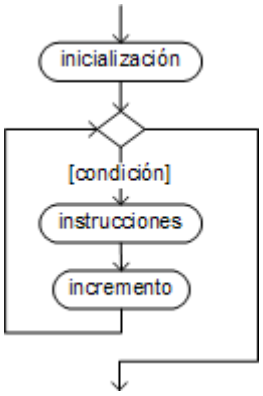
6.1. Lógica de la estructura For

La lógica general de esta estructura es la siguiente:



Donde la *acción* se repite desde que la variable de control tiene un valor inicial *(vi)* hasta que alcanza o supera un valor final *(vf)*. En cada repetición la variable de control es incrementada (o disminuida) en un determinado valor *(incr)*.

Aun cuando en general la lógica de la estructura “for” es la descrita en el párrafo anterior, esta estructura es la menos estándar de todas y su implementación varía de lenguaje a lenguaje, así en “hecl” la lógica es la siguiente:



Y como se puede ver esta estructura se parece más a la estructura “while” que a la tradicional estructura “for”. En hecl, la “inicialización” no está limitada a una sentencia de asignación, es válida cualquier instrucción e inclusive es posible escribir dos o más instrucciones. La “condición” tiene un comportamiento similar a la estructura “while”, es decir el ciclo se repite mientras dicha condición es verdadera. El “incremento” no está limitado tampoco a un incremento propiamente, sino que en el mismo se puede escribir cualquier instrucción o inclusive dos o más instrucciones.

Al igual que en la estructura “while”, en la estructura “for” se pueden emplear los modificadores “break” (para salir del ciclo), “continue” (para

saltar al siguiente ciclo) y “return” (para salir directamente del ciclo y devolver un resultado).

La sintaxis de esta estructura en hecl es la siguiente:

for {inicialización} {condición} {incremento} {instrucciones}

En consecuencia la estructura “for” de hecl (al igual que la de C) es muy flexible y puede emplearse no sólo en ciclos donde se conoce el número de iteraciones, sino en muy diversas situaciones, sin embargo, en lo posible procuraremos emplear la misma sobre todo en la forma tradicional, es decir cuando se conoce el número de veces que debe repetirse el ciclo.

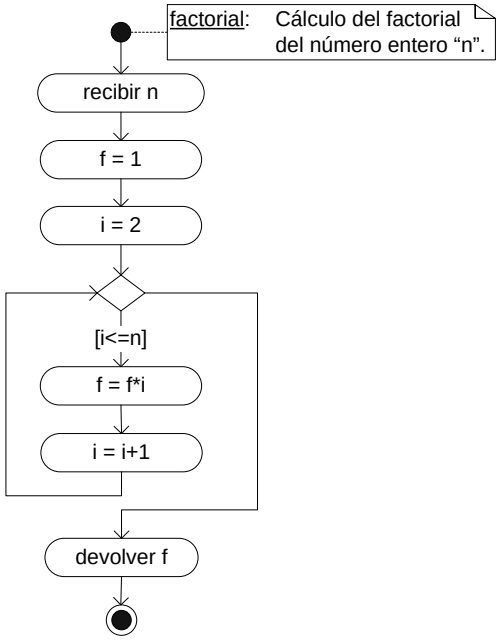
6.2. Ejemplos

1. Cálculo del factorial

Como primer ejemplo, elaboraremos un módulo para calcular el factorial de un número entero, aplicando la fórmula de definición:

$$n! = 1*2*3*...*n = \prod_{i=1}^{n} i; \qquad 0! = 1 \tag{6.1}$$

Este es un problema clásico que puede ser resuelto fácilmente con la estructura for, pues al tratarse de una productoria finita, se sabe el número de veces que debe repetirse el proceso. Así por ejemplo para calcular el factorial de 5, siendo el valor inicial 1, en la primera repetición se multiplica ese número por 2: 1*2=2, en la segunda se multiplica el resultado anterior por 3: 2*3=6, en la tercera por 4: 6*4=24 y en la cuarta por 5: 24*5=120, siendo este resultado (120) el factorial de 5. En consecuencia para calcular el factorial de 5 el proceso debe repetirse 4 veces, variando el contador desde 2 hasta 5. De manera similar, para calcular el factorial de cualquier número “n” el proceso se repite el proceso “n-1” veces, desde 2 hasta “n”:



El código respectivo es el siguiente:

```
proc factorial n {
   if {!= [% $n 1] 0} {throw "El número debe ser entero"}
   if {< $n 0} {throw "El número debe ser mayor o igual a cero"}
   set f 1
   for {set i 2} {<= $i $n} {incr $i 1} {set f [* $f $i]}
   return $f
}
```

Haciendo correr el programa con algunos valores se obtiene:

```
hecl> factorial 3
6
hecl> factorial 5
120
hecl> factorial 7
5040
hecl> factorial 10
3628800
hecl> factorial 12
479001600
```

Sin embargo cuando se hace correr el programa con 20 el resultado es:

```
hecl> factorial 20
-2102132736
```

Que por supuesto es erróneo. Esto sucede no porque exista un error en el código o en la lógica, sino porque se está trabajando con valores enteros y el límite de los valores enteros positivos es 2147483647 y el factorial de 20 supera ampliamente este límite (de hecho ya el factorial de 13 supera este límite). Por ello, y a pesar de que se sabe que el resultado es entero, resulta de mayor utilidad trabajar en el programa con números reales. Para ello simplemente se inicializa f en 1. en lugar de 1:

```
proc factorial n {
   if {!= [% $n 1] 0} {throw "El número debe ser entero"}
   if {< $n 0} {throw "El número debe ser mayor o igual a cero"}
   set f 1.
   for {set i 2} {<= $i $n} {incr $i 1} {set f [* $f $i]}
   return $f
}
```

Con esta simple modificación podemos calcular ahora el factorial de 20 e inclusive de números mayores:

```
hecl> factorial 5
120.0
hecl> factorial 20
2.43290200817664E18
hecl> factorial 50
3.0414093201713376E64
```

Por supuesto el resultado sólo tiene los 16 o 17 dígitos de precisión con los que se trabaja en hecl cuando se realizan operaciones con números reales.

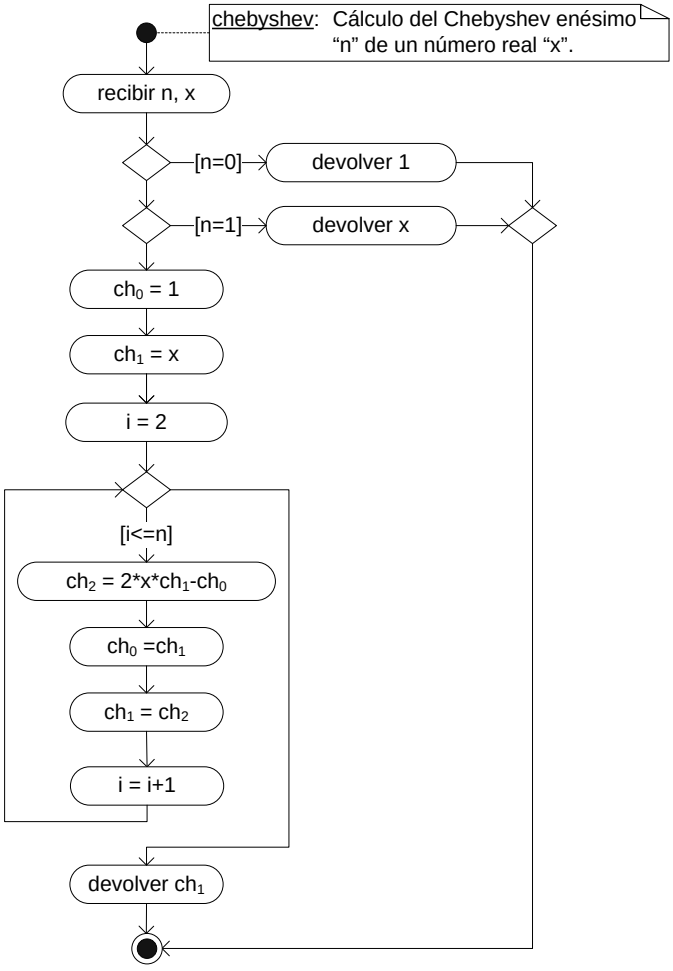
2. Cálculo del Chebyshev

Como segundo ejemplo elaboraremos un módulo para calcular el chebyshev enésimo de un número real, empleando su fórmula de definición:

$$Ch_{n+1}(x) = 2xCh_n(x) - Ch_{n-1}(x)$$
$$Ch_0(x) = 1$$
$$Ch_1(x) = x$$

La lógica para resolver este problema es básicamente la misma que para el Fibonacci: Comenzando con los valores conocidos (Ch_0 y Ch_1) se calcula el tercero (Ch_2), entonces con los dos últimos valores calculados (Ch_1 y Ch_2) se calcula el cuarto y así sucesivamente hasta llegar al Chebyshev que se quiere calcular. En otras palabras, el proceso debe repetirse "n-1" veces: desde 2 hasta n, por ello, como se sabe el número de veces que debe repetirse el proceso, resulta más adecuado resolver el problema con la estructura "for".

La lógica, en forma de diagrama de actividades, es la siguiente:



Y el código respectivo es:

```
proc chebyshev { n x } {
  if {!= [% $n 1] 0} {throw "El órden debe ser entero"}
  if {< $n 0} {throw "El órden debe ser mayor o igual a cero"}
  if {= $n 0} {return 1}
  if {= $n 1} {return $x}
  set ch0 1.
  set ch1 $x
  for {set i 2} {<= $i $n} {incr $i 1} {
    set ch2 [- [* 2 $x $ch1] $ch0]
    set ch0 $ch1
    set ch1 $ch2
  }
  return $ch1
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
chebyshev 4 4.36
2739.8423372800007
hecl> chebyshev 3 7.34
1559.767616
hecl> chebyshev 10 15.1
3.80094721711099956E14
hecl> chebyshev 7 20.2
8.745310747609 84E10
hecl> chebyshev 9 0.45
-0.8720108055
```

Resultados que pueden ser corroborados con datos tabulados o empleando otra aplicación como Mathematica.

3. Cálculo del coseno

Como tercer ejemplo emplearemos la estructura “for” para calcular el coseno de un ángulo en radianes empleando la serie de Taylor:

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ...\infty, \quad -\infty < x < \infty$$

En este caso emplearemos la estructura “for” no en su forma tradicional, sino aprovechando la flexibilidad que tiene en hecl. En consecuencia, la lógica es la misma que para resolver cualquier serie: a) inicializar variables; b) calcular el nuevo término; c) calcular el nuevo valor de la sumatoria; d) comparar las dos últimos sumatorias y si son aproximadamente iguales el proceso concluye siendo la respuesta la última sumatoria calculada, caso contrario se intercambia variables, se incrementa contadores y se repite el proceso desde el paso (b).

Prácticamente la única variación importante (aparte de las condiciones límite y excepciones) es la forma en que se calcula el nuevo término. Así en esta serie los nuevos términos se calculan multiplicando el término anterior por $-x^2$ y dividiendo entre dos valores consecutivos. Por ejemplo para calcular el cuarto término, en base al tercero, la operación es:

$$\left(\frac{x^4}{4!}\right)\left(\frac{-x^2}{5*6}\right)=-\frac{x^6}{6!}$$

Como ya se dijo, cuando una serie, como la del presente ejemplo, involucra restas de números muy grandes o muy pequeños se pierden definitivamente dígitos. Por ello siempre que sea posible se debe tratar de trabajar sólo con sumas o si ello no es posible con valores relativamente pequeños.

En este caso no es posible evitar las restas, por lo que debemos buscar la forma de trabajar con números pequeños y ello se logra reduciendo los ángulos mayores a 2π a su equivalente comprendido entre 0 y 2π. Para ello recordamos que en el coseno (al igual que en el seno) se repiten los valores entre 0 y 2π sin importar cuantas vueltas se dé al cuadrante, por lo tanto el resultado es el mismo si se emplea el número completo o sólo la fracción de la última vuelta al cuadrante:

$$\text{Ángulo comprendido entre } 0 \text{ y } 2\pi = \text{Parte fraccionaria de}\left(\frac{\text{Ángulo original}}{2\pi}\right)*2\pi$$

Así por ejemplo se obtiene el mismo resultado calculando el coseno de 3483.47 que calculando el coseno de:

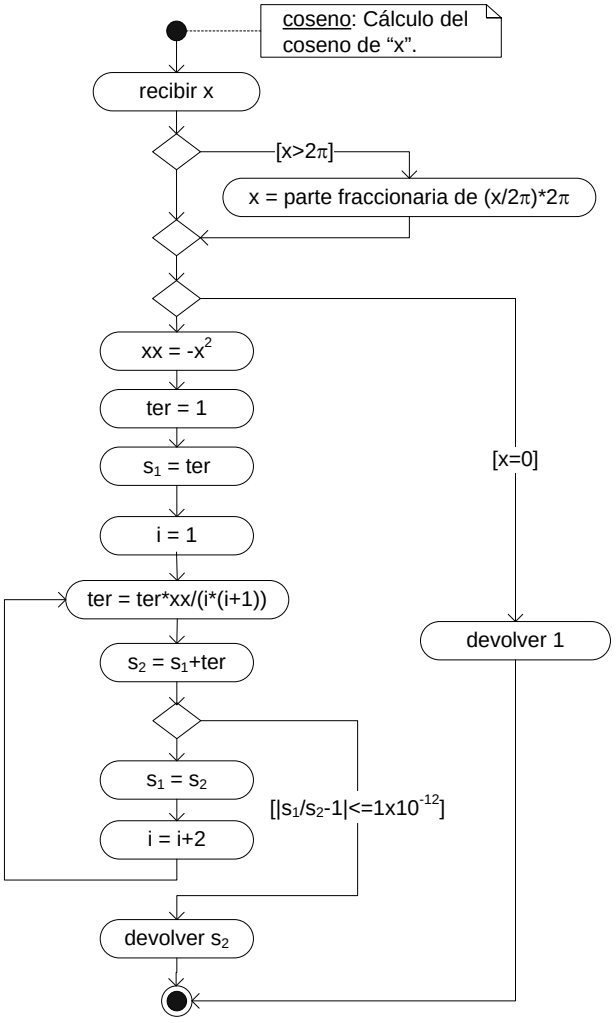
$$\text{Parte fraccionaria de}\left(\frac{3483.47}{2\pi}\right)*2\pi = 2.58533982047$$

Tomando en cuenta las anteriores consideraciones se ha elaborado la lógica que se presenta en el diagrama de actividades en la siguiente página. El código elaborado en base al mismo es el siguiente:

```
proc coseno x {
  set PI 3.1415926535897793
  if {> $x [* 2 $PI]} {set x [* [% [/ $x [* 2 $PI]] 1][* 2 $PI]]}
  if {= $x 0} {return 1}
  set xx [* -1 $x $x]
  set ter 1.
  set s1 $ter
  for {set i 1.} true {set s1 $s2; set i [+ $i 2]} {
    set ter [* $ter [/ $xx [* $i [+ $i 1]]]]
    set s2 [+ $s1 $ter]
    if {< [abs [- [/ $s1 $s2] 1]] 1e-15} {return $s2}
  }
}
```

Haciendo correr el programa con algunos valores de prueba y corroborando los mismos con la función “cos” de hecl se obtiene:

```
hecl> coseno 0.5
0.8775825618903728
hecl> cos 0.5
0.8775825618903728
hecl> coseno -2.1
-0.5048461045998575
```




```
hecl> cos -2.1
-0.5048461045998576
hecl> coseno 4.5
-0.21079579943078058
hecl> cos 4.5
-0.2107957994307797
hecl> coseno 12.3
0.9647326178866054
hecl> cos 12.3
0.9647326178866098
hecl> coseno 34.5
-0.9983462274487429
hecl> cos 34.5
-0.9983462274487422
```

Por supuesto es posible hacer consideraciones adicionales aparte de “x=0”, pues se sabe el valor de coseno para otros valores como π/2, 3π/2, π, etc. En este caso no obstante se ha preferido no hacer dichas consideraciones para no perder de vista el objetivo principal, que es el resolver la serie.

4. Integración numérica Método de Simpson

Como cuarto ejemplo elaboraremos un módulo para calcular el valor de integrales definidas con el método de Simpson:

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f(x_1) + 4\sum_{i=2,4,6}^{n} f(x_i) + 2\sum_{i=3,5,7}^{n-1} f(x_i) + f(x_{n+1}) \right)$$

$$h = \frac{b-a}{n}$$

Donde "f" es la función a integrar, "x_1" es el límite inferior ("a"), "x_{n+1}" es el límite superior ("b"), *"n"* es el número de segmentos que existe entre "a" y "b" (el cual debe ser par) y "h" es el ancho de cada uno de esos segmentos.

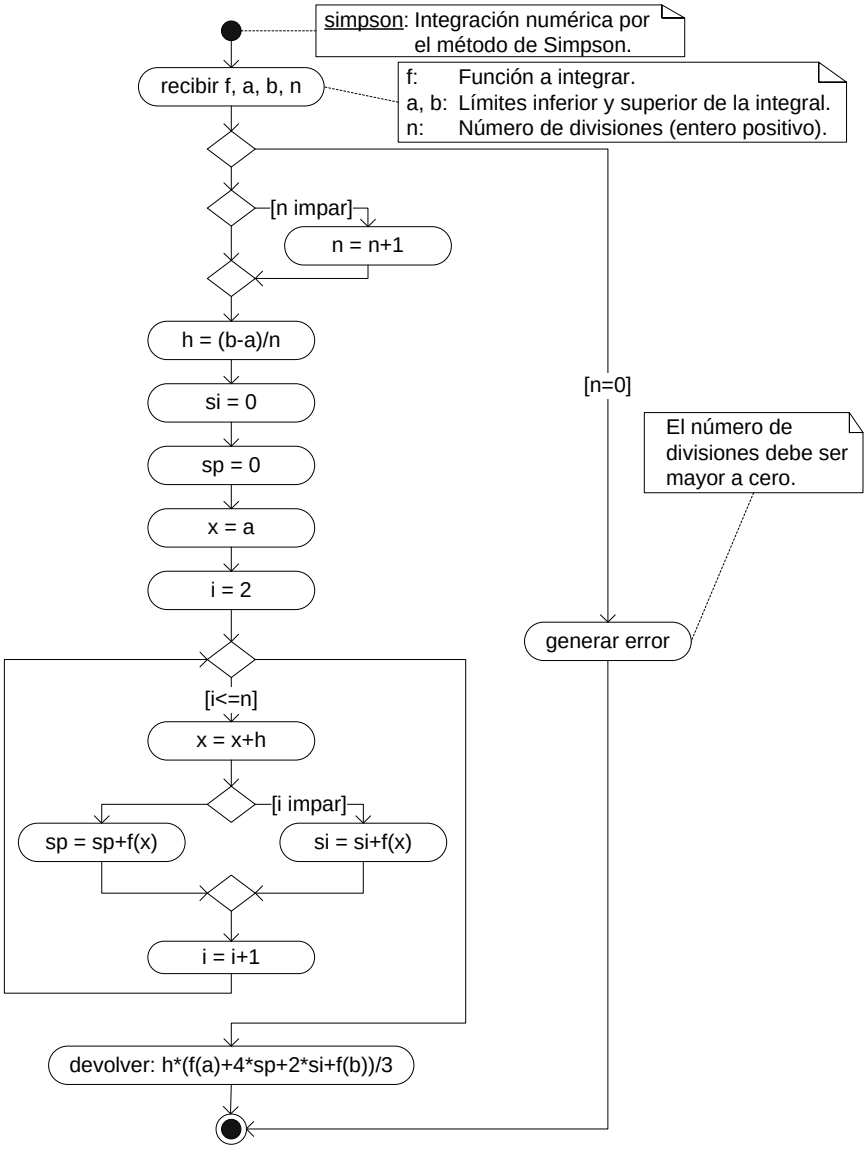
Los valores de "x_2", "x_3", etc., se calculan sumando al valor anterior "h", así "x_2" es igual a "x_1+h", "x_3" es igual a "x_2+h", "x_4" es igual a "x_3+h" y así sucesivamente.

Las dos sumatorias corresponden simplemente a un ciclo "For" que va desde 2 hasta n, donde cuando el número es par se guarda la sumatoria en una variable y cuando es impar en otra.

El número de divisiones "n", con el cual se calcula el ancho de cada segmento "h", es el que determina la exactitud con la que se obtiene el resultado, cuando mayor es el valor de "n" (y en consecuencia menor el valor de "h") mayor es la precisión del resultado obtenido. Por supuesto un valor elevado de "n" implica un mayor número de iteraciones y un valor muy pequeño de "h" puede resultar en un resultado menos exacto debido a los errores de redondeo.

El algoritmo para el método de Simpson se muestra en la siguiente página y como se puede ver en el mismo, antes de comenzar los cálculos se pregunta si el número de divisiones "n" es impar y de ser así se le suma "1" para que se convierta en un número par, pues como se dijo, para el método de Simpson "n" siempre debe ser par.

```
proc simpson {f a b n} {
  if {!= [% $n 1] 0} {throw "El N° de divisiones debe ser entero"}
  if {= $n 0} {throw "El N° de divisiones debe ser mayor a 0"}
  if {> $a $b} {throw "El lím. inferior debe ser menor al superior"}
  if {!= [% $n 2] 0} {incr $n}
  set h [/ [- $b $a] [* $n 1.]]
  set si 0.
  set sp 0.
  set x $a
  for {set i 2} {<= $i $n} {incr $i} {
    set x [+ $x $h]
    set fx [upeval [list $f $x]]
    if {!= [% $i 2] 0} {
      set si [+ $si $fx]} else {set sp [+ $sp $fx]}
  }
  set fa [upeval [list $f $a]]
  set fb [upeval [list $f $b]]
  return [/ [* $h [+ $fa [* 4 $sp][* 2 $si] $fb]] 3]
}
```



Podemos probar el programa calculando por ejemplo la siguiente integral:

$$\int_1^3 (x^3 + 2x^2 + 3x + 4)dx$$

Para ello primero programamos la función:

```
proc fx x { + [* $x $x $x][* 2 $x $x][* 3 $x] 4 }
```

Y llamamos al programa "simpson" con dicha función, los límites y un número de divisiones, por ejemplo 20:

```
hecl> simpson fx 1 3 20
57.333333333333338
```

Para verificar que el número de divisiones elegido es adecuado, volvemos a llamar "Simpson" con un número mayor de divisiones, digamos 80:

```
hecl> simpson fx 1 3 40
57.33333333333331
```

Puesto que el resultado en ambos casos es esencialmente el mismo, entonces podemos asumir que la solución, con el número de divisiones elegido, es la correcta (podemos corroborar este resultado integrando manualmente la ecuación o empleando alguna otra herramienta como Mathematica).

Simplemente para demostrar que la función elaborada permite integrar cualquier función de una variable, emplearemos el mismo para calcular la integral:

$$\int_{10}^{20}\left(\frac{3x^2+2x-4}{5x^3-4x^{2.1}}\right)dx$$

Igual que en el caso anterior, primero programamos la función:

```
proc f2 x {
   set num [+ [* 3 $x $x][* 2 $x] -4]
   set den [- [* 5 $x $x $x][* 4 [pow $x 2.1]]]
   / $num $den
}
```

Y llamamos a “Simpson” con digamos 40 divisiones:

```
hecl> simpson f2 10 20 40
0.46809467562258167
```

Verificamos si el número de divisiones es adecuado volviendo a llamar a “Simpson” con 80 divisiones:

```
hecl> simpson f2 10 20 80
0.4680946641945484
```

Como se puede observar ambos resultados son iguales en los primeros 7 dígitos, por lo que podemos asumir que el resultado es preciso hasta ese número de dígitos. Simplemente para verificar que es así volvemos a llamar a “Simpson” con 160 divisiones:

```
hecl> simpson f2 10 20 160
0.4680946634792687
```

Y ahora los dos últimos valores calculados son iguales en 8 dígitos, por lo que este último resultado debe ser preciso al menos en ese número de dígitos (algo que una vez más puede ser corroborado con Mathematica).

6.3. Ejercicios

1. Elabore el diagrama de flujo y el código de un módulo que devuelva 10 elevado al número de dígitos de los que consta un número entero positivo o negativo.

2. Elabore el diagrama de flujo y el código de un módulo que calcule la sumatoria de los números impares que existen entre 1 y un número dado “n”: $\sum_{i=1,3,5}^{n} i$.

3. Elabore el diagrama de flujo y el código de un módulo que calcule la productoria de los números pares que existen entre 1 y un número dado "n": $\prod_{i=2,4,6}^{n} i$.

4. Elabore el diagrama de flujo y el código de un módulo que calcule el valor de la productoria: $\prod_{i=2,4,6}^{n} (x+i)$, pruebe el módulo con "n=10, x=3.2" y "n=20, x=20.3".

5. Elabore el diagrama de flujo y el código de un módulo que calcule el Legendre enésimo de un número real "x", con la ecuación: $(n+1)Le_{n+1}(x)-(2n+1)xLe_n(x)+nLe_{n-1}(x)=0$ donde por definición: $Le_0(x)=1$, $Le_1(x)=x$.

6. Elabore el diagrama de flujo y el código de un módulo que calcule, con 9 dígitos de precisión, el seno de un ángulo con la serie: $sen(x)=x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+...\infty, \quad -\infty<x<\infty$, donde el ángulo está en radianes. En el módulo se deben convertir los ángulos mayores a 2π a su equivalente comprendido entre 0 y 2π.

7. Elabore el diagrama de flujo y el código de un módulo que calcule, con 15 dígitos de precisión, la raíz cúbica de un número real "n" con la fórmula de Newton: $x_2=(2x_1+n/x_1^2)/3$.

8. Elabore el diagrama de flujo el código de un módulo que calcule el arco tangente hiperbólico de un número real empleando la serie de Taylor: $\tanh^{-1}(x) = x+x^3/3+x^5/5+x^7/7+ ...\infty;\ |x|<1$.

9. Elabore el diagrama de flujo y el código de un módulo para integrar funciones con una incógnita por el método del trapecio:

$$\int_a^b f(x)dx \approx \frac{h}{2}\left(f(x_1)+2\sum_{i=2}^{n} f(x_i)+f(x_{n+1})\right); \quad x_1=a \ ; x_{n+1}=b; \ h=\frac{b-a}{n}$$

Pruebe el método resolviendo la siguiente integral:

$$\int_{1.3}^{8.4}\left(\frac{\sqrt{2x^{4.1}+3x^3-2x^{1.45}}}{\sqrt[3]{6.2x^{3.7}-2x^{2.7}+4}}\right)dx$$

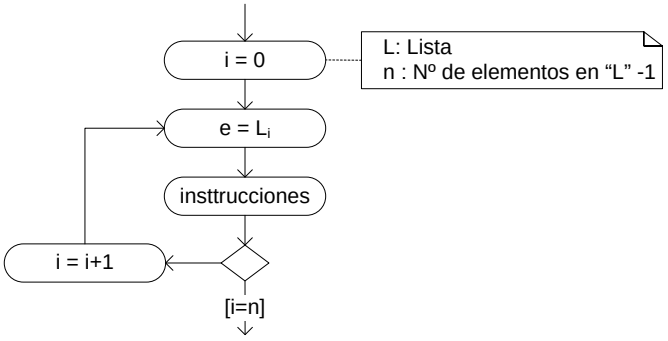
7. ITERACIÓN - 3

En ocasiones se requiere realizar operaciones con un conjunto de elementos en lugar valores aislados. Si bien ello es posible empleando las estructuras ya estudiadas, especialmente la estructura “for”, en hecl contamos con una instrucción más específica para este fin: la instrucción “foreach”, la cual puede resultar más eficiente en algunos casos.

En este tema estudiaremos algunos ejemplos para acostumbrarnos al uso de esta instrucción. El objetivo es el de comprender la forma de funcionamiento de la instrucción “foreach” para luego poder aplicarla en aquellos casos donde facilite y/o clarifique la codificación.

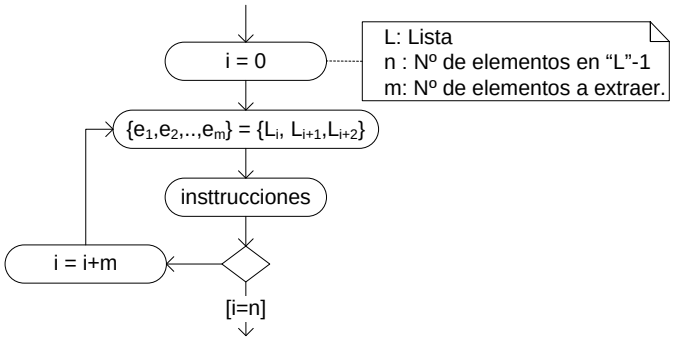
7.1. Lógica de la estructura foreach

La lógica de la estructura “foreach” es aproximadamente la siguiente:



Como se puede ver esta instrucción extrae en cada repetición del ciclo uno de los elementos de la lista, el cual es asignado a una variable y permite realizar operaciones con la misma dentro del bloque de instrucciones. Observe que en el diagrama de actividades el contador comienza en cero, ello se debe a que en “hecl” (como en “c” y otros lenguajes) el índice del primer elemento es cero. Si bien en esta instrucción en particular no se tiene acceso al índice, es conveniente irse acostumbrando a la forma en que operan los índices dentro de una lista en “hecl”.

Alternativamente es posible extraer dos o más elementos al mismo tiempo, en cuyo caso la lógica es la siguiente:



En este caso se extraen dos o más elementos en las variables especificadas en la lista y luego se pueden hacer operaciones con las mismas dentro del bloque de instrucciones. Un detalle que se debe tomar en cuenta cuando se extraen dos o más elementos es que el número de elementos en la lista debe ser, necesariamente, un múltiplo del número de elementos a extraer, así

por ejemplo, si en cada repetición del ciclo se extraen 3 elementos, entonces la lista podrá tener 3, 6, 9, 12, 15, .., etc. elementos, pues de lo contrario se produce un error.

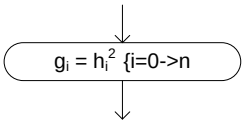
La sintaxis de esta estructura en “hecl” para el primer caso es la siguiente:

foreach variable lista {instrucciones}

Y la siguiente para el segundo caso:

foreach {variables} lista {instrucciones}

En la práctica la lógica suele expresarse como instrucciones simples que operan sobre listas en lugar de valores simples, y no como se ha mostrado en los diagramas anteriores. Por ejemplo para calcular el cuadrado de todos los elementos de una lista y asignarlos a otra se escribe una instrucción como la siguiente:



Pues la codificación en sí depende del lenguaje, así en el caso de “hecl”, si la lista está conformada por los números del 1 al 5, la instrucción anterior se codificaría de la siguiente forma:

```
hecl> set h {1 2 3 4 5}
1 2 3 4 5
hecl> set g {}
hecl> foreach e $h {lappend $g [* $e $e]}
```

Donde los valores calculados y almacenados en la variable “h” son:

```
hecl> puts $g
1 4 9 16 25
```

En el anterior ejemplo se ha empleado la instrucción “lappend”, la misma que añade al final de una lista uno o más valores. Su sintaxis es la siguiente:

lappend lista valor

Donde “lista” es la lista o variable que contiene una lista y “valor” es el valor o valores (separados con espacios) que se añaden al final de la lista.

Como la instrucción “foreach” trabaja con listas, veremos además, algunas de las instrucciones que nos permiten trabajar con ellas. Ya sabemos que para crear una lista podemos escribir directamente los elementos entre llaves (separados por espacios) o emplear la instrucción “list”:

list valores

Así por ejemplo para crear la lista “l1” con los números 3 4 5 6 8 10, podemos emplear indistintamente cualquiera de las siguientes instrucciones:

```
hecl> set l1 {3 4 5 6 8 10}
3 4 5 6 8 10
hecl> set l1 [list 3 4 5 6 8 10]
3 4 5 6 8 10
```

Para extraer uno de los elementos de una lista se emplea la instrucción “lindex”, que tiene la siguiente sintaxis:

lindex lista índice

Donde lista es la lista o variable que contiene una lista e índice es el número de elemento que se quiere extraer, recordando que en “hecl” el primer índice es cero. Por ejemplo para extraer el 4 elemento de la lista “l1” escribimos:

```
hecl> lindex $l1 3
6
```

La instrucción que en cierto sentido es el complemento de la anterior es “lset”, la cual permite reemplazar un valor de la lista con otro. Su formato es el siguiente:

lset lista índice valor

Donde “lista” es la lista o variable que contiene una lista, “índice” es la posición del elemento que será reemplazado y “valor” es el valor de reemplazo. Por ejemplo para reemplazar el quinto elemento de la lista “l1” por 20.2 escribimos:

```
hecl> lset $l1 4 20.2
3 4 5 6 20.2 10
```

Si no se escribe ningún valor, entonces el elemento de lista referenciado es eliminado, por ejemplo para eliminar el primer elemento de la lista “l1” escribimos:

```
hecl> lset $l1 0
4 5 6 20.2 10
```

Para averiguar el número de elementos de una lista se emplea la instrucción “llen”:

llen lista

Así si aplicamos esta instrucción a la lista “l1” obtenemos:

```
hecl> llen $l1
5
```

Para extraer un conjunto de elementos consecutivos empleamos la instrucción “lrange”:

lrange lista índice_inicial índice_final

Por ejemplo, para extraer los elementos 2, 3 y 4 de la lista “l1” escribimos:

```
hecl> lrange $l1 1 3
5 6 20.2
```

Emplearemos algunas de estas instrucciones en los ejemplos de este tema y los volveremos a emplear más tarde cuando estudiemos de manera más formal los datos estructurados.

7.2. Ejemplos

1. Sumatoria de una lista de valores

Como primer ejemplo, elaboraremos un módulo para calcular la sumatoria de los elementos de una lista de números.

Si la lista es “l”, básicamente lo que se tiene que hacer para resolver el problema es:

$$\sum_{i}^{n} l_i \quad \{i = 0 \rightarrow n-1$$

Puesto que la lógica es muy simple escribiremos directamente el código:

```
proc sum l {
   set s 0
   foreach x $l {set s [+ $s $x]}
   return $s
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> sum {1 2 3 4 5 6 7 8 9 10}
55
hecl> sum {}
0
hecl> sum {3 8 9 1 4 6 2}
33
```

2. Calcular el valor de una expresión con cada uno de los elementos de una lista

Como segundo ejemplo calcularemos el valor de la siguiente expresión con cada uno de los elementos de una lista y devolveremos los resultados en la lista original:

$$x^3+2x^2+3x\ +4\ \ \{i=0\rightarrow n-1$$

Una vez más la lógica es muy sencilla: simplemente se aplica el valor de la anterior expresión a cada uno de los elementos de la lista y se guardan y devuelven los resultados en la lista recibida. Es decir aplicamos la siguiente ecuación:

$$x_i^3+2x_i^2+3x_i+4\ \ \{i=0\rightarrow n-1$$

El código que resuelve el problema es el siguiente:

```
proc ejem2 l {
   set i 0
   foreach x $l {
      lset $l $i [+ [* $x $x $x][* 2 $x $x][* 3 $x] 4]
      incr $i
   }
   return $l
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> ejem2 {1 2 3 4}
10 26 58 112
hecl> ejem2 {1.1 1.2 1.3 1.4 1.5 1.6 1.7}
11.051000000000002 12.207999999999998 13.477 14.863999999999
997 16.375 18.016000000000002 19.793
```

Si creamos una variable con una lista, como la siguiente:

```
hecl> set r {1.1 1.2 1.3 1.4 1.5 1.6}
1.1 1.2 1.3 1.4 1.5 1.6
```

Y mandamos esta lista al programa obtenemos:

```
hecl> ejem2 $r
11.051000000000002 12.207999999999998 13.477 14.863999999999
997 16.375 18.016000000000002
```

Ahora si imprimimos el valor de la variable “r” obtenemos:

```
hecl> puts $r
11.051000000000002 12.207999999999998 13.477 14.863999999999
997 16.375 18.016000000000002
```

Donde como vemos se han reemplazado los valores de lista original con los nuevos valores calculados, no obstante que en ninguna parte del código se ha empleado el comando “upeval”. Ello se debe a que en “hecl”, como ocurre en muchos otros lenguajes, las matrices se mandan por referencia y no por valor, por lo tanto cualquier cambio que se hace en la matriz dentro del módulo implica en realidad un cambio en la matriz original.

Este es un factor que se debe tomar muy en cuenta cuando se trabaja con listas pues es una de las causas más frecuente de errores.

Para comprender mejor este hecho veamos que pasa por ejemplo cuando se crea la variable “a” con una lista de números:

```
hecl> set a {1 2 3 4 5}
1 2 3 4 5
```

Y se asigna luego esta lista a la variable “b”:

```
hecl> set b $a
1 2 3 4 5
```

Verificamos el contenido de la variable “b”:

```
hecl> puts $b
1 2 3 4 5
```

Ahora cambiamos el valor del tercer elemento de la lista contenida en “b” por 4.56:

```
hecl> lset $b 2 4.56
1 2 4.56 4 5
```

Aparentemente hasta este punto todo está bien, no obstante, si vemos imprimimos el valor de la variable “a”:

```
hecl> puts $a
1 2 4.56 4 5
```

Nos llevamos la sorpresa de que en la misma también ha cambiado el valor del tercer elemento, no obstante que no hemos hecho ninguna operación con la variable “a”. Una vez más ello se debe a que cuando se trabaja con listas se trabaja por referencia, por lo tanto lo que se ha asignado a la variable “b” no son los elementos de la lista, sino la posición de memoria donde se encuentra la lista, por lo tanto cualquier cambio que se haga a la variable “b” es también un cambio a la variable “a” o viceversa, porque en realidad ambas variables tienen la misma dirección de memoria y en consecuencia son la misma lista.

3. Factorial de un número

Como tercer ejemplo elaboraremos el código de un módulo para calcular el factorial de un número de una forma un tanto diferente a la estudiada en el anterior tema.

En este caso la lógica a emplear será la siguiente: se crea una lista con números consecutivos desde 2 hasta el número cuyo factorial se quiere calcular; luego se multiplican los elementos de esta lista. Por supuesto esta no es la forma más eficiente de resolver el problema, pero es una forma que nos permite aplicar y comprender mejor la instrucción “foreach”.

El código respectivo es el siguiente:

```
proc fac n {
    if {!= [% $n 1] 0} {throw "El número debe ser entero"}
```

```
    if {< $n 0} {throw "El número debe ser positivo"}
    set l { }
    for {set i 2} {<= $i $n} {set i [+ $i 1]} {lappend $l $i}
    set f 1.
    foreach x $l {set f [* $f $x]}
    return $f
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> fac 5
120.0
hecl> fac 9
362880.0
hecl> fac 30
2.6525285981219103E32
```

Observe que para incrementar el valor del contador “i” en el ciclo “for” se ha empleado el operador de suma y no el procedimiento “incr”. Si se emplea este “incr” en lugar de “i”, es decir si el programa se modifica de la siguiente forma:

```
proc fac n {
    if {!= [% $n 1] 0} {throw "El número debe ser entero"}
    if {< $n 0} {throw "El número debe ser positivo"}
    set l { }
    for {set i 2} {<= $i $n} {incr $i} {lappend $l $i}
    set f 1.
    foreach x $l {set f [* $f $x]}
    return $f
}
```

Al calcular el factorial de 5 se obtiene:

```
hecl> fac 5
1296.0
```

Esto también se debe a que el procedimiento “incr” trabaja con referencias, por lo tanto cuando se añaden elementos a la lista “l” en realidad sólo se añade la referencia a la variable “i” y al final en la lista quedan “n-1” valores con el mismo número (en el ejemplo 6).

4. Raíz cuadrada

Como cuarto ejemplo modificaremos el programa que calcula la raíz cuadrada de manera que pueda trabajar tanto con datos simples como con listas.

Por supuesto la fórmula que permite el cálculo de la raíz cuadrada sigue siendo la misma:

$$x_2 = \frac{1}{2}\left(x_1 + \frac{n}{x_1} \right)$$

Y la lógica también, por lo que no se volverá a explicar la misma. Lo único que cambia es que ahora el proceso se repite tantas veces como elementos tenga la lista recibida.

Y el código respectivo es:

```
proc rcuad l {
  set r {}
  foreach n $l {
    if {or [= $n 0][= $n 1]} {lappend $r $n; continue}
    set x [abs $n]
    if {> $x 0} {set x1 [/ 2. $x]} else {set x1 [* 2. $x]}
    while true {
      set x2 [/ [+ $x1 [/ $x $x1]] 2.]
      if {< [abs [- [/ $x1 $x2] 1]] 1e-16} break
      set x1 $x2
    }
    if {< $n 0} {lappend $r [list 0 $x2]} else {lappend $r $x2}
  }
  return $r
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> rcuad 2.1
1.449137674618944
hecl> rcuad {2 3 4 5}
1.414213562373095 1.7320508075688772 2.0 2.23606797749979
hecl> rcuad {4.5 6.7 9.9}
2.121320343559643 2.588435821108957 3.1464265445104544
hecl> rcuad {-4.5 -2.3 1.1 2.2}
{0 2.121320343559643} {0 1.51657508881031} 1.048808848170151
6 1.4832396974191326
```

Como se puede observar, el programa devuelve ahora resultados tanto cuando se le manda un valor simple como cuando se le manda una lista, y como ya se implementó en el tema 5, devuelve también resultados imaginarios.

5. Arco Tangente Hiperbólico

Como quinto ejemplo elaboraremos un módulo que calcule el arco tangente hiperbólico tanto de número simples como de listas empleando la serie de Taylor:

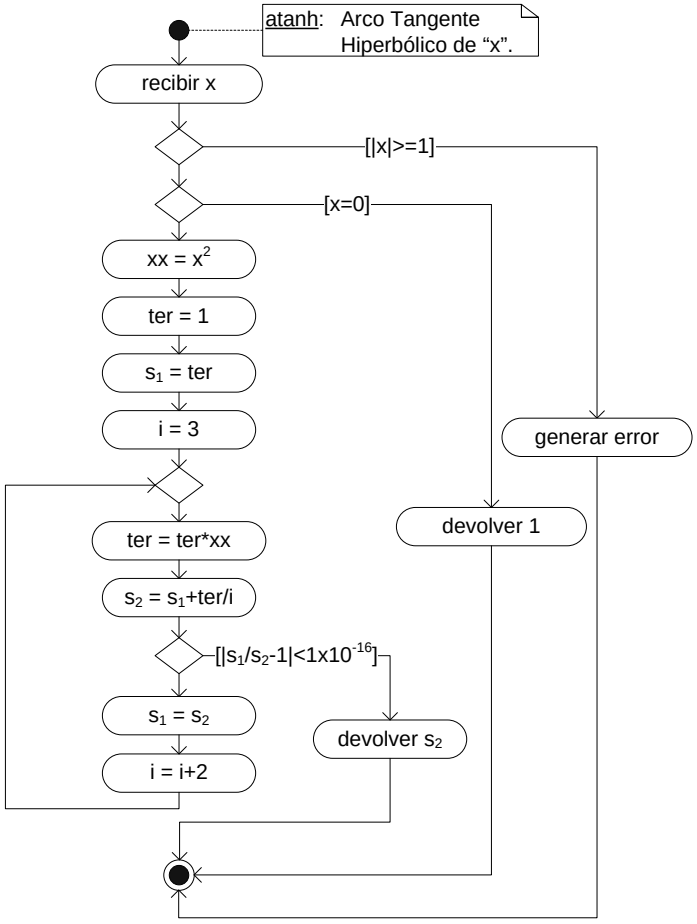
$$tanh^{-1}(x) = 1 + \frac{x^3}{3} + \frac{x^5}{5} + \frac{x^7}{7} + ...\infty, \quad |x| < 1$$

La lógica para calcular el arco tangente hiperbólico es esencialmente la misma que la de cualquier serie. Dicha lógica será analizada por separado y luego simplemente la misma se repetirá (igual que en el ejemplo anterior), el número de elementos que tenga la lista recibida.

Como casi siempre ocurre con las series, la única variación importante (aparte de las condiciones límite y excepciones) es la forma en que se calcula el nuevo término. En esta serie los términos son básicamente los numeradores, pues los denominadores son simplemente números impares, los cuales se pueden obtener fácilmente con un contador. Ahora, como se pude ver, el nuevo numerador se obtienen simplemente multiplicando el anterior por x^2. Así para obtener el cuarto numerador a partir del tercero realizamos la operación:

$$\left(\frac{x^5}{}\right)\left(\frac{x^2}{}\right) = \frac{x^7}{}$$

Como ya se dijo, en este caso, al calcular el nuevo término no se toma en cuenta el denominador, pues el mismo es simplemente un contador. El algoritmo para el cálculo del arco tangente hiperbólico de un número real es el siguiente:



Para trabajar con listas, simplemente se repite este algoritmo tantas veces como elementos tenga la lista y en lugar de devolver directamente los resultados, se van añadiendo a la lista resultante.

El código respectivo es el siguiente:

```
proc atanh l {
  set r {}
  foreach x $l {
    if {>= [abs $x] 1} {throw "El número debe ser menor a 1"}
    if {= $x 0} {lappend $r 1; continue}
    set xx [* $x $x]
    set ter $x
    set s1 $ter
    for {set i 3.} true {set i [+ $i 2]; set s1 $s2} {
      set ter [* $ter $xx]
      set s2 [+ $s1 [/ $ter $i]]
      if {< [abs [- [/ $s1 $s2] 1]] 1e-16} {lappend $r $s2; break}
    }
  }
```

```
    return $r
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> atanh 0.34
0.3540925289622429
hecl> atanh (0.1 0.34 0.45 0.67 0.89)
0.10033534773107556 0.3540925289622429 0.48470027859405174 0.
8107431254751376 1.4219258711306353
hecl> atanh (-0.3 -0.51 -0.75)
-0.3095196042031116 -0.5627297693521487 -0.9729550745276568
hecl> atanh (0.34 0.64 1.23)
(ERROR (El n·mero debe ser menor a 1)) (throw 1) (if 2) (for
        at org.hecl.Interp.run(Unknown Source)
El n·mero debe ser menor a 1
```

Que como se puede comprobar con otro programa (o calculadora) devuelve los resultados correctos.

7.3. Ejercicios

1. Elabore un módulo que calcule la productoria de los elementos de una lista de números.

2. Elabore un módulo que calcule la sumatoria de los cuadrados de los elementos de una lista de números.

3. Elabore un módulo que reciba una lista de números y devuelva una lista con las raíces cuadradas de cada uno de los elementos de la lista.

4. Elabore un módulo que reciba una lista de números y devuelva una lista donde con los resultados de la siguiente expresión para cada uno de los elementos de la lista.

$$\sqrt[3]{\frac{\ln(x+3)}{e^{x+5}}}$$

5. Elabore un módulo que empleando el comando “foreach” calcule el Fibonacci de un número entero: $F_n = F_{n-1}+F_{n-2}$; $F_1 = F_2 = 1$.

6. Elabore un módulo que calcule, con 15 dígitos de precisión, la raíz cúbica tanto de números simples como de listas, con la fórmula de Newton: $x_2=(2x_1+n/x_1^2)/3$.

7. Elabore un módulo que calcule, con 16 dígitos de precisión, el seno tanto de un ángulo simple como de una lista de ángulos, empleando la serie de Taylor: $sen(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...\infty, \quad -\infty < x < \infty$, donde el ángulo está en radianes. En el módulo se deben convertir los ángulos mayores a 2π a su equivalente comprendido entre 0 y 2π.

8. RECURSIVIDAD

En este tema estudiaremos otro método que podemos emplear para resolver problemas iterativos: la recursividad.

La recursividad no es imprescindible para resolver problemas iterativos, sin embargo, en ocasiones, los problemas pueden ser resueltos de manera más clara y sencilla empleando este método.

Puesto que una solución más sencilla es también más entendible, el empleo de estos métodos puede facilitar el mantenimiento y actualización de los programas. Recordemos que la programación estructura ha surgido justamente con la intención de facilitar el mantenimiento y actualización de programas, por lo tanto cualquier método o técnica que facilite la elaboración y mantenimiento de programas debe ser tomado en cuenta y utilizado si cumple con ese propósito.

En consecuencia el objetivo del presente tema es que al concluir el mismo estén capacitados para resolver problemas iterativos aplicando un razonamiento recursivo.

8.1. ANÁLISIS RECURSIVO

En general se dice que un término o concepto es recursivo si en su definición se emplea el término o concepto que se está definiendo.

Aplicado a la programación estructurada, se dice que ***un módulo es recursivo si en su solución se llama a sí mismo***.

La recursividad implica sobre todo otra forma de pensar: ***cuando se lleva a cabo el análisis recursivo de un problema se piensa en cómo obtener el resultado aprovechando otro resultado***.

Así por ejemplo, para calcular el factorial de un número, aplicando el razonamiento recursivo, se piensa en la manera de calcular el factorial aprovechando un resultado previo y para ese fin recordamos que el factorial también se define como:

$$n! = (n-1)!*n$$

$$0! = 1$$

Así el factorial de 5 se calcula multiplicando el factorial de 4 por 5 (5! = 4!*5 = 24*5 = 120), por lo tanto podemos calcular el factorial de un número simplemente multiplicando el factorial del número anterior por dicho número. Ahora bien, cuando se piensa de manera recursiva es muy importante no pensar en todo el proceso, sino sólo en la solución recursiva. Por ejemplo, en el cálculo del factorial, no pensamos en cómo se calcula el factorial del número anterior (de eso se encarga el proceso recursivo), simplemente nos aseguramos que es posible calcular el nuevo factorial multiplicando el factorial del número anterior por el nuevo número, si es posible, entonces ya tenemos la solución recursiva correcta.

Algo que siempre debe aparecer en una solución recursiva es la condición de finalización, es decir una condición para la cual se conoce el resultado. Por ejemplo en el caso del factorial se sabe que el factorial de 0 es 1, por lo tanto esta constituye la condición de finalización. **En las soluciones recursivas es importante no olvidar nunca la condición de finalización**, pues su omisión da lugar a ciclos infinitos, lo que en la práctica se traduce en un desbordamiento de la memoria.

Puesto que la mejor forma de aprender es mediante el ejemplo, se resuelven a continuación algunos ejercicios empleando el razonamiento recursivo.

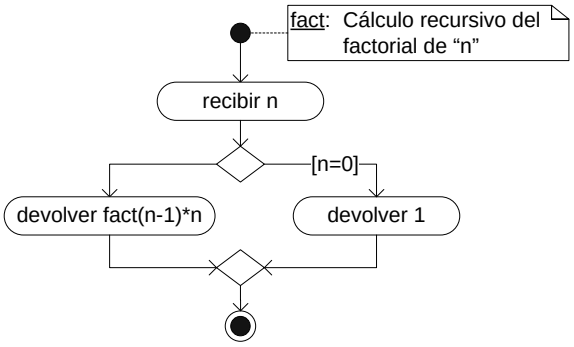
8.2. EJEMPLOS

1. Cálculo del factorial

Comenzaremos con el ejemplo clásico del factorial, cuya fórmula recursiva fue dada previamente:

$$n! = (n-1)! * n$$
$$0! = 1$$

Como ya se explicó, la solución recursiva es muy sencilla, lo único que se debe hacer es programar esa expresión. Y aunque la solución es muy sencilla, se presenta a continuación el diagrama de actividades respectivo:



El código respectivo es el siguiente:

```
proc fact n {
   if {= $n 0} {return 1.} else {* [fact [- $n 1]] $n}
}
```

Que evidentemente es más sencillo que la solución no recursiva, no obstante, desde el punto de vista de la eficiencia, las soluciones recursivas son siempre menos eficientes pues consumen más tiempo y memoria que las soluciones no recursivas. En realidad **la única razón por la cual se emplea la recursividad es porque simplifica la solución de algunos problemas iterativos**.

Haciendo correr el módulo con algunos valores de prueba se obtiene:

```
hecl> fact 5
120.0
hecl> fact 7
5040.0
hecl> fact 50
3.041409320171337 6E64
```

Que son las soluciones correctas.

Antes de pasar a analizar algunos otros aspectos de esta solución es conveniente analizar primero como opera la recursividad. Si bien en la práctica se recomienda no pensar en dichos detalles, el comprender el funcionamiento recursivo ayuda a aprovechar mejora la recursividad.

Veamos entonces qué pasa por ejemplo cuando calculamos el factorial de 5. En la primera llamada (no recursiva) a *fact* el parámetro *n* toma el valor 5 y se ejecuta la instrucción:

** [fact [- $n 1]] $n = * [fact 4] 5*

Ahora, para calcular el factorial de 4 (fact 4) el módulo se llama a si mismo y en esta llamada, dado que "n" es mayor a 0, se ejecuta la instrucción:

** [fact [- $n 1]] $n = * [fact 3] 4*

Una vez más, para calcular el factorial de 3, el módulo se llama a si mismo y el proceso se repite hasta que "n" es igual a 1:

** [fact [- $n 1]] $n = * [fact 2] 3*

** [fact [- $n 1]] $n = * [fact 1] 2*

** [fact [- $n 1]] $n = * [fact 0] 1*

Entonces el módulo se llama a si mismo con "n=0", y como para el mismo se sabe el resultado (1), el mismo es devuelto al módulo que hizo la llamada (en este caso el módulo con "n=1"), el cual emplea dicho resultado para devolver a su vez un resultado:

** [fact 0] 1 = * 1 1 = 1*

Este resultado es devuelto al módulo que hizo la llamada (el módulo con n=2) y así sucesivamente hasta llegar al módulo que hace la primera llamada recursiva:

** [fact 1] 2 = * 1 2 = 2*

** [fact 2] 3 = * 2 3 = 6*

** [fact 3] 4 = * 6 4 = 24*

** [fact 4] 5 = * 24 5 = 120*

Siendo este el resultado final devuelto por el módulo recursivo.

Como se puede observar el proceso es moroso, sin embargo de ello se encarga automáticamente el lenguaje, lo único que se debe hacer para que el proceso se lleve a cabo es escribir la instrucción recursiva. Por otra parte se hace notar que cada vez que un módulo se llama a si mismo se crea un nuevo conjunto de variables para dicho módulo y cuando termina se liberar la memoria reservada. Estas actividades consumen tanto tiempo como memoria siendo la razón por la cual las soluciones recursivas son menos eficientes que las soluciones iterativas.

Se recalca que se ha mostrado como opera el proceso recursivo sólo para una mejor comprensión de la recursividad, pero que en la práctica, cuando se resuelven problemas recursivos, se debe evitar en pensar en estos detalles. La recursividad constituye esencialmente una forma más sencilla de resolver algunos problemas iterativos y es más sencilla justamente porque no es necesario preocuparse de los detalles internos.

Volvamos ahora al problema del factorial, el cual en realidad ya ha sido resuelto, sin embargo, en esta solución no se han tomado en cuenta los casos especiales: como cuando el número es real (en lugar de entero) o cuando el número es negativo.

Es importante comprender que dichas comprobaciones **no deben ser escritas en el módulo recursivo**, porque de ser así, como el módulo se llama a sí mismo "n" veces, dichas comprobaciones se harían "n" veces, algo del todo ilógico, pues sólo se requiere una comprobación.

Por ello, el módulo recursivo se mantiene tal como ha sido escrito previamente y lo que se hace es crear otro módulo, donde se hacen las comprobaciones necesarias y desde el cual se llama al módulo recursivo. En

el caso del factorial, los dos módulos son los que se muestran a continuación, donde se ha cambiado el nombre del módulo recursivo a facr, simplemente para que el módulo principal, desde el cual se llama al módulo recursivo, tenga el nombre “fact”:

```
proc factr n {
   if {= $n 0} {return 1.} else {* [factr [- $n 1]] $n}
}

proc fact n {
   if {!= [% $n 1] 0} {throw "El número debe ser entero"}
   if {< $n 0} {throw "El número debe ser positivo"}
   factr $n
}
```

Ahora haciendo correr el programa se obtienen los mismos resultados que antes, pero además cuando se manda números reales o menores a 0 se obtienen los errores correspondientes:

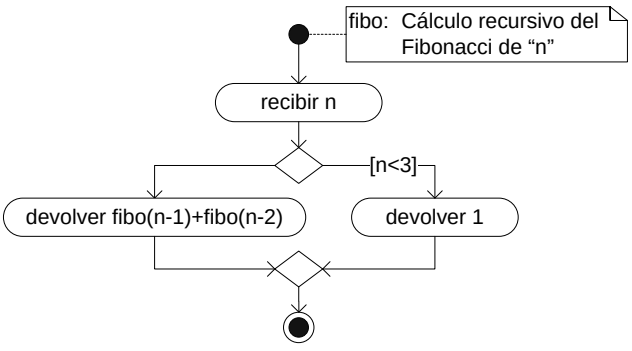
```
hecl> fact 5
120.0
hecl> fact -5
(ERROR (El n·mero debe ser positivo)) (throw 1) (if 3) (fact
        at org.hecl.Interp.run(Unknown Source)
El n·mero debe ser positivo
hecl> fact 8.2
(ERROR (El n·mero debe ser entero)) (throw 1) (if 2) (fact 1
        at org.hecl.Interp.run(Unknown Source)
El n·mero debe ser entero
```

2. Calculo del Fibonacci

Como segundo ejemplo elaboraremos un módulo para el cálculo del Fibonacci, recordando que la fórmula de definición del mismo es:

$$F_n = F_{n-1} + F_{n-2}; \quad F_1 = F_2 = 1$$

Que como se ve, es de hecho una definición recursiva, pues se calcula un nuevo Fibonacci en base a otros dos valores conocidos. Aparentemente entonces la solución es muy sencilla y directa:



Siendo el código respectivo (sin comprobar condiciones límite) el siguiente:

```
proc fibo n {
   if {< $n 3} {return 1.} else {+ [fibo [- $n 1]][fibo [- $n 2]]}
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> fibo 8
21.0
hecl> fibo 10
55.0
hecl> fibo 15
610.0
```

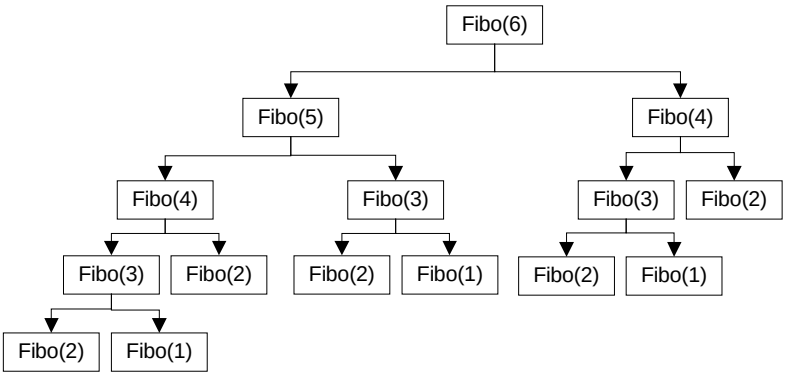
Que son los valores correctos, por lo que podríamos concluir que el módulo es correcto, sin embargo, si hacemos calculamos el Fibonacci de 20, notaremos que demora un poco en devolver la respuesta:

```
hecl> fibo 20
6765.0
```

Y cuando calculamos el Fibonacci de 30, tenemos que esperar varios segundos antes de obtener una respuesta:

```
hecl> fibo 30
832040.0
```

Entonces advertimos que algo no está tan bien en el módulo elaborado. Para comprender el por qué la lógica es errónea, analicemos las llamadas recursivas que se llevan a cabo para calcular el *Fibonacci* de 6:



Como se puede observar, para calcular el Fibonacci de 6 se realizan 3 llamadas recursivas para calcular el Fibonacci de 1; 5 llamadas recursivas para calcular el Fibonacci de 2; 3 llamadas para el Fibonacci de 3; 2 llamadas para el Fibonacci de 4 y una llamada para el Fibonacci de 5. Es decir un total de 14 llamadas recursivas, en las cuales se vuelven a calcular, de manera ilógica, valores que ya fueron calculados previamente. Si la lógica fuera correcta serían suficientes un máximo de 6 llamadas recursivas (no 14).

Este exceso en llamadas recursivas incrementa geométricamente con el número del Fibonacci. Así por ejemplo para calcular el Fibonacci de 15 se requiere 1218 llamadas recursivas (en lugar de 15), para calcular el Fibonacci de 17 son necesarias 3192 llamadas recursivas, para calcular el Fibonacci de 30 se requieren 1664078 llamadas recursivas y para calcular el Fibonacci de 34, 11405772 llamadas recursivas: ¡más de 11 millones de llamadas por encima de las necesarias!, algo absolutamente ilógico e ineficiente.

En estos casos resulta que a pesar de ser la fórmula de definición recursiva, la solución más sencilla y eficientes es no recursiva. Esto es algo que ocurre casi siempre cuando en la solución recursiva el módulo debe llamarse a si mismo dos o más veces, razón por la cual en esos casos se debe tener especial cuidado para evitar módulos ineficientes e ilógicos como el anterior.

Para lograr una solución recursiva que sea práctica, es necesario emplear algún mecanismo que permita almacenar y recuperar valores previamente calculados, evitando así el volver a calcular los mismos valores una y

otra vez de manera irracional. Algo que no se debe permitir nunca sin importar que el mismo sea o no recursivo.

Una alternativa consiste en el empleo de una variable global (una lista), en la cual se van almacenando los valores del Fibonacci a medida que son calculados y del cual se recuperan los valores previamente calculados en lugar de volver a calcularlos. La implementación de dicha alternativa es la siguiente:

```
set f {1. 1.}
proc fibor n {
  global f
  set ne [llen $f]
  if {<= $n $ne} {return [lindex $f [- $n 1]]} else {
    set r [+ [fibor [- $n 1]][fibor [- $n 2]]]
    lappend $f $r
    return $r
  }
}
```

Como se ve, en esta implementación la variable global es “f”, la cual se inicializa con los dos primeros valores conocidos del Fibonacci: 1 y 1. Esta variable se declarada como “global” al interior del procedimiento y si el número cuyo Fibonacci se quiere calcular es menor o igual al número de elementos de la misma (ne), se extrae el resultado directamente de la misma, caso contrario se hacen las llamadas recursivas y todo nuevo resultado calculado se añade a la lista “f” de manera que no vuelva a ser calculado una y otra vez.

Es obvio que con las anteriores correcciones la lógica deja de ser tan simple como en la solución directa, pero si no se procede de esta manera la lógica sería errónea. La otra alternativa por supuesto es emplear la lógica iterativa en lugar de la recursiva.

Como ocurrió con el anterior ejemplo, las comprobaciones adicionales como las del valor inferior permitido y la del tipo de dato se hacen en un procedimiento aparte, desde el cual se llama al módulo recursivo:

```
proc fibo n {
  if {!= [% $n 1] 0} {throw "El número debe ser entero"}
  if {< $n 2} {throw "El número debe ser positivo"}
  fibor $n
}
```

Haciendo correr el módulo con algunos valores de prueba se obtienen los resultados correctos:

```
hecl> fibo 1
1.0
hecl> fibo 5
5.0
hecl> fibo 8
21.0
hecl> fibo 15
610.0
hecl> fibo 20
6765.0
```

Pero ahora el cálculo del factorial de 30 no demora como ocurría con el anterior módulo:

```
hecl> fibo 30
832040.0
```

Y es perfectamente posible calcular el Fibonacci de números aún mayores, por ejemplo el Fibonacci de 100:

```
hecl> fibo 100
3.54224848179262E20
```

Ahora, si mostramos el valor de la variable "f" (la lista), veremos que contiene todos los Fibonacci desde 1 hasta 100:

```
hecl> puts $f
1.0 1.0 2.0 3.0 5.0 8.0 13.0 21.0 34.0 55.0 89.0 144.0 233.0
 377.0 610.0 987.0 1597.0 2584.0 4181.0 6765.0 10946.0 17711
.0 28657.0 46368.0 75025.0 121393.0 196418.0 317811.0 514229
.0 832040.0 1346269.0 2178309.0 3524578.0 5702887.0 9227465.
0 1.4930352E7 2.4157817E7 3.9088169E7 6.3245986E7 1.02334155
E8 1.65580141E8 2.67914296E8 4.33494437E8 7.01408733E8 1.134
90317E9 1.836311903E9 2.971215073E9 4.807526976E9 7.77874204
9E9 1.2586269025E10 2.0365011074E10 3.29512800 99E10 5.331629
1173E10 8.6267571272E10 1.39583862445E11 2.25851433717E11 3.
65435296162E11 5.91286729879E11 9.56722026041E11 1.548008755
92E12 2.504730781961E12 4.052739537881E12 6.557470319842E12
1.0610209857723E13 1.7167680177565E13 2.7777890035288E13 4.4
945570212853E13 7.2723460248141E13 1.176690304609 94E14 1.903
92490709135E14 3.080615211701 29E14 4.98454011879264E14 8.065
15533049393E14 1.304969544928657E15 2.111485077978 05E15 3.41
6454622906707E15 5.527939700884757E15 8.944394323791464E15 1
.447233402467622E16 2.341672834846768 4E16 3.7889062373143904
E16 6.1305790721611584E16 9.919485309475548 8E16 1.6050064381
6367072E17 2.5969549691112256E17 4.201961407274 8966E17 6.798
9163763861222E17 1.1000877783661018 9E18 1.7799794160047 1398E
18 2.8800671943708 16E18 4.6600466103755305E18 7.540113804746
3465E18 1.2200160415121877E19 1.974027421986 8226E19 3.194043
46349901E19 5.1680708854858 33E19 8.362114348984843E19 1.3530
185234470676E20 2.189229958345552E20 3.54224848179262E20
```

En estos casos, más que los anteriores, es evidente que una solución iterativa sería más eficiente, pero de elegirse una solución recursiva, debe procederse de manera similar a como se ha mostrado en este ejemplo.

3. Potencia Entera de un Número Real

Como tercer ejemplo elaboraremos un módulo para calcular la potencia entera de un número real: x^n.

Desde el punto de vista recursivo el problema se resuelve multiplicando por "*x*" la potencia de x^{n-1}, es decir:

$$x^n = x^{n-1} * x$$

Y puesto que cualquier número elevado a uno *(n=1)* es igual al mismo número, esta puede ser la condición de finalización. No obstante, esta solución aunque matemáticamente correcta, no es eficiente. Por ejemplo para calcular $3.45^{2500000}$, se deben realizar 2 millones quinientas mil llamadas recursivas e igual número de multiplicaciones.

Una solución más eficiente se consigue tomando en cuenta que x^n puede ser calculado con $(x^{n/2})^2$ si *n* es par y $(x^{(n-1)/2})^2 * x$ si *n* es impar.

Así por ejemplo el valor de x^{4561} puede ser calculado con:

$$x^{4561} = (x^{(4561-1)/2})^2 * x = (x^{2280})^2 * x$$

Para calcular x^{2280} aplicamos el mismo procedimiento:

$$x^{2280} = (x^{2280/2})^2 = (x^{1140})^2$$

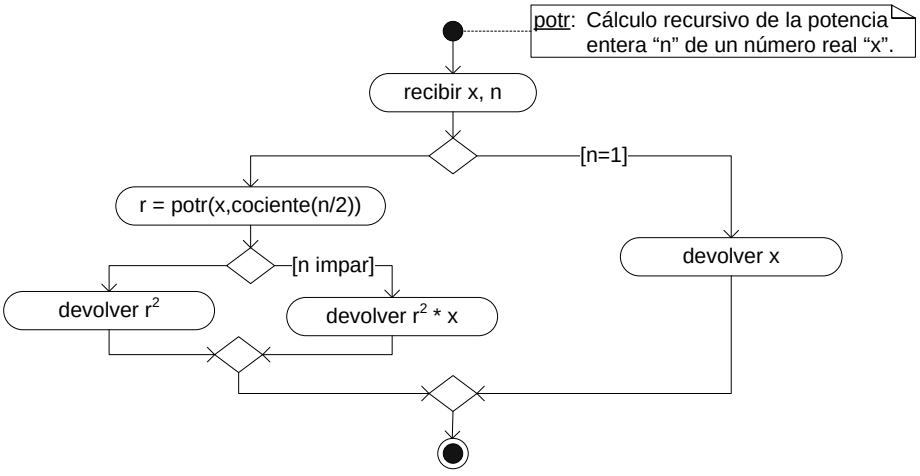
Prosiguiendo de esta manera los siguientes cálculos recursivos serían:

$$x^{1140} = (x^{1140/2})^2 = (x^{570})^2$$
$$x^{570} = (x^{570/2})^2 = (x^{285})^2$$
$$x^{285} = (x^{(285-1)/2})^2 * x = (x^{142})^2 * x$$
$$x^{142} = (x^{142/2})^2 = (x^{71})^2$$
$$x^{71} = (x^{71/2})^2 * x = (x^{35})^2 * x$$
$$x^{35} = (x^{(35-1)/2})^2 * x = (x^{17})^2 * x$$
$$x^{17} = (x^{(17-1)/2})^2 * x = (x^{8})^2 * x$$
$$x^{8} = (x^{8/2})^2 = (x^{4})^2$$
$$x^{4} = (x^{4/2})^2 = (x^{2})^2$$
$$x^{2} = (x^{2/2})^2 = (x^{1})^2$$
$$x^{1} = x$$

Con este procedimiento se consigue un considerable ahorro de tiempo y recursos, así en lugar de las 4561 llamadas recursivas (y 4561 multiplicaciones) sólo son necesarias 13 llamadas recursivas y 18 multiplicaciones.

Como se puede observar este planteamiento es por naturaleza recursivo, de manera que puede ser implementado con mayor facilidad aprovechando la recursividad.

El algoritmo del módulo recursivo, que es donde realmente se resuelve el problema, es el siguiente:



Y el código respectivo es:

```
proc potr {x n} {
  if {= $n 1} {return $x} else {
    set r [potr $x [/ $n 2]]
    if {!= [% $n 2] 0} {return [* $r $r $x]} else {return [* $r $r]}
  }
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> potr 2.1 7
180.10885410000003
hecl> potr 7.9 300
1.9414552611813962E269
```

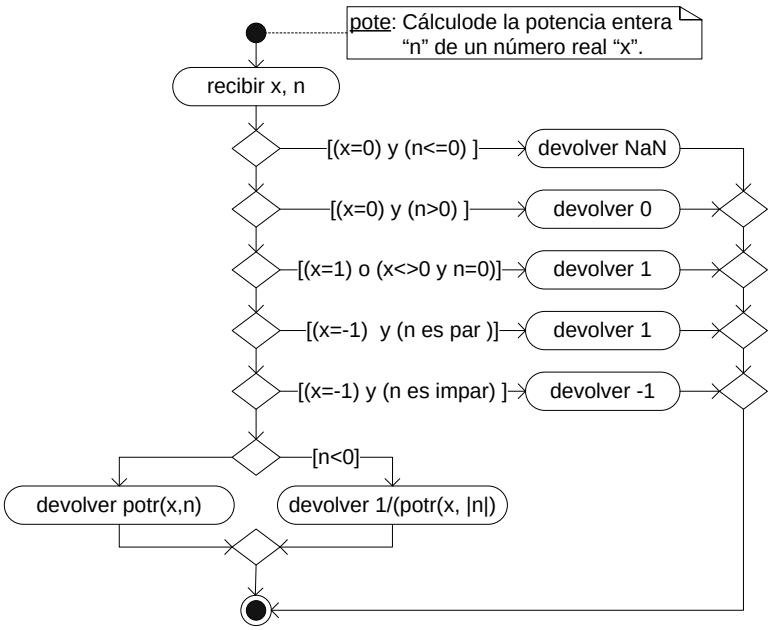
```
hecl> potr 2.45 500
3.8286192822287354E194
```

Valores que pueden ser verificados por ejemplo con “pow”.

Por supuesto en el módulo recursivo no se han tomado en cuenta los muchos casos que tienen la potencia entera de un número real: en ocasiones (como cuando la base es 1) se sabe el resultado sin necesidad de hacer ningún cálculo y en otros (como cuando la base es 0 y la potencia 0) el resultado es indefinido, por lo que en la mayoría de estos casos el procedimiento recursivo daría resultados erróneos o quedaría en un ciclo infinito. Tampoco se controlan los tipos de datos, así el módulo recursivo queda en un ciclo infinito si se le manda una potencia real.

Al igual que en los anteriores ejemplos, todas estas condiciones se analizan en un módulo separado (desde el cual se llama al módulo recursivo), esto, como ya se dijo, para evitar que estas condiciones sean verificadas una y otra vez (de forma irracional) en cada llamada al ciclo.

El algoritmo de dicho módulo es el siguiente:



Y el código respectivo es:

```
proc pote {x n} {
   if {!= [% $n 1] 0} {throw "La potencia debe ser entera"}
   if {and [= $x 0][<= $n 0]} {return NaN
   } elseif {and [= $x 0][> $n 0]} {return 0
   } elseif {or [= $x 1][= $n 0]} {return 1
   } elseif {and [= $x -1][= [% $n 2] 0]} {return 1
   } elseif {and [= $x -1][!= [% $n 2] 0]} {return -1
   } elseif {< $n 0} {return [/ 1 [potr $x [abs $n]]]
   } else {return [potr $x $n]}
}
```

Haciendo correr el módulo con algunos valores de prueba se obtiene:

```
hecl> pote 5.2 6.7
{ERROR {La potencia debe ser entera}} {throw 1} {if 2} {pote
 1}
```

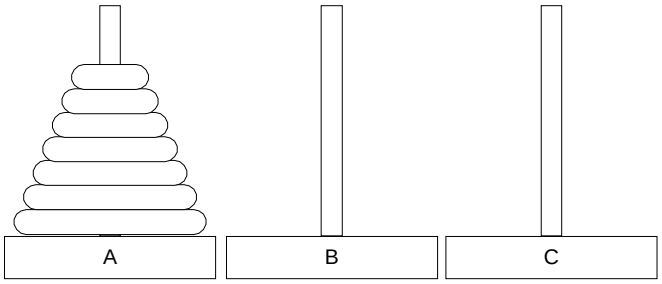
```
La potencia debe ser entera
hecl> pote 0 0
NaN
hecl> pote 0 45
0
hecl> pote 1 89
1
hecl> pote 34.32 0
1
hecl> pote -1 5
-1
hecl> pote 2.1 7
180.10885410000003
hecl> pote 4.5 -32
1.2508336587712131E-21
hecl> pow 4.5 -32
1.2508336587712131E-21
```

Y como era de esperar se siguen obteniendo los resultados correctos (pues el módulo que los obtiene en realidad no cambia), pero donde además se ha tomando en cuenta el tipo de dato y los posibles casos.

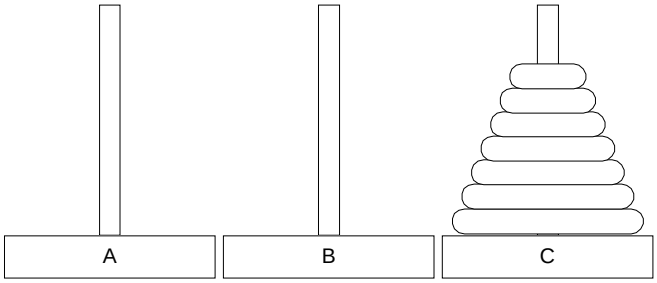
4. Las Torres de Hanói

Como cuarto ejemplo elaboraremos un módulo que resuelva el juego de las torres de Hanói, devolviendo los movimientos necesarios para resolver el mismo.

Este juego que consta de tres postes y un número determinado aros de diferente diámetro. Al inicio del juego los aros se encuentran ordenados en el primer poste "A", tal como se muestra en la figura:

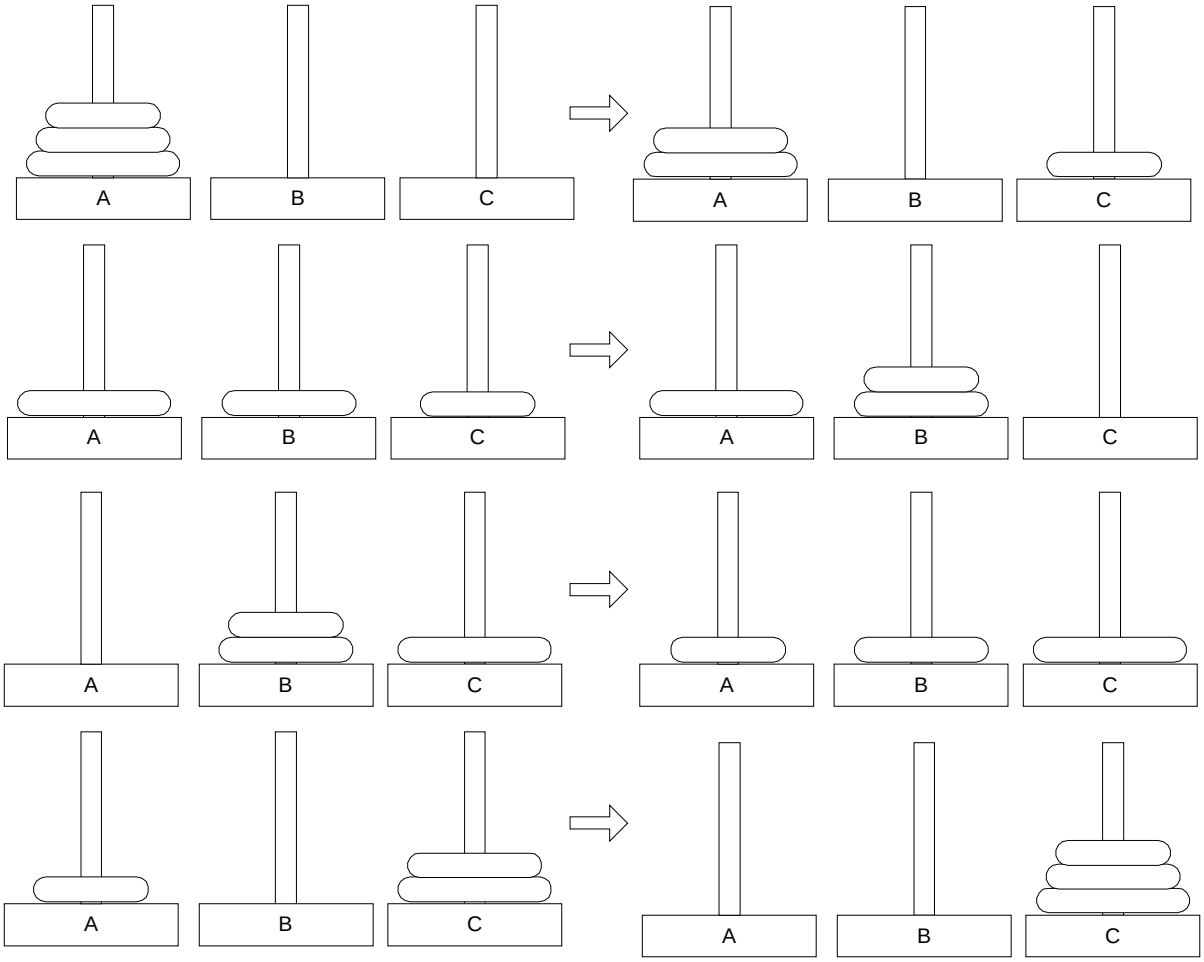


El objetivo del juego es hacer que al final estos aros queden ordenados en el tercer poste, tal como se muestra en la figura:



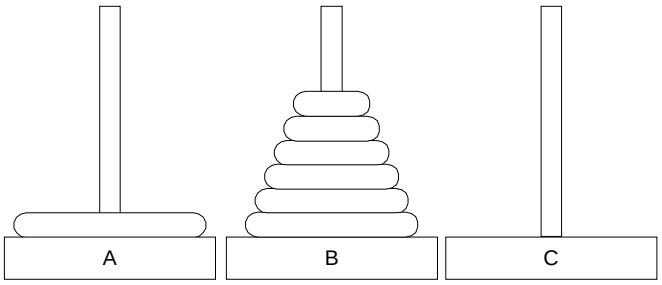
Para conseguir este objetivo, se debe mover un aro a la vez, empleando también el poste auxiliar "B", pero cuidando de que nunca un aro de menor diámetro quede debajo de uno de mayor diámetro.

Por ejemplo si el juego consta de tres aros, se deben realizar los movimientos que se detallan en las siguientes figuras, para resolver el juego:

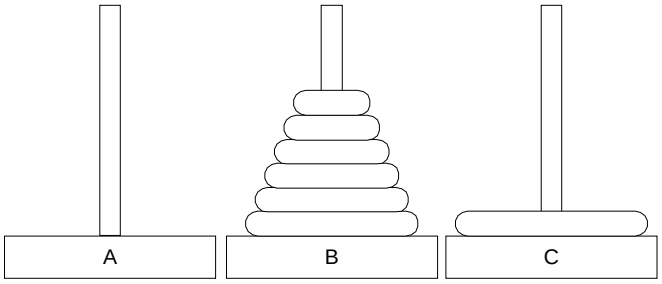


Es decir que en este caso los movimientos necesarios para ordenar los aros en el poste "C" son: "A->C", "A->B", "C->B", "A->C", "B->A", "B->C" y "A->C", que se entiende como: mover el último aro del poste "A" al poste "C", luego mover el último aro del poste "A" al poste "B", después mover el último aro del poste "C" al poste "B", mover el último aro del poste "A" al poste "C", mover el último aro del poste "B" al poste "A", mover el último aro del poste "B" al poste "C" y finalmente mover el último aro del poste "A" al poste "C".

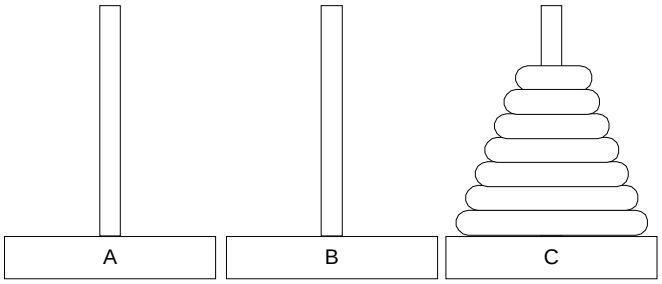
Este es un problema por naturaleza recursivo, por lo tanto la solución es mucho más sencilla siguiendo un razonamiento recursivo: para llevar "n" aros desde un poste de origen a otro de destino, debemos mover "n-1" aros del poste de origen al poste auxiliar (aquel que no es ni el origen ni el destino):



Luego movemos el último aro del poste de origen al poste de destino:



Finalmente movemos los "n-1" aros del poste auxiliar al poste de destino:



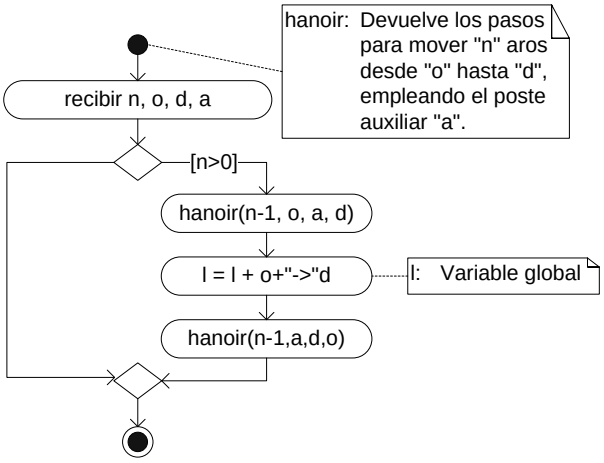
Y listo, ¡con estos tres movimientos el problema está resuelto! y sin importar el número de aros que se tenga.

En cierto sentido se puede pensar que se haciendo trampa, porque aparentemente no se están siguiendo las reglas del juego, pero no es así, lo único que no se está haciendo es preocuparse por los pasos intermedios, es decir en esta solución no importa cómo se mueve cada uno de los “n-1” discos del poste “A” al poste “B”, ni como se mueven luego estos mismos “n-1” discos del poste “B” al poste “C”. Pero eso es algo que justamente nos permite la recursividad: el no preocuparse de los detalles. Para mover esos “n-1” discos el módulo se llamará a si mismo cuantas veces sea necesario, pero de eso se encarga el proceso recursivo. El único trabajo que debe hacer el programador es cerciorarse que el razonamiento recursivo produce el resultado deseado (y en este caso vemos que es así).

Sin embargo, para que el razonamiento recursivo esté completo se requiere además una condición de finalización. Para ello simplemente observamos que si un poste se queda sin aros, entonces no quedan más movimientos que hacer y el proceso concluye par dicho aro. Esta será entonces la condición de finalización: si el poste queda sin aros, el proceso concluye para el mismo.

Con las anteriores consideraciones, el algoritmo para mover “n” aros desde un poste hasta otro, lo que implica como ya se ha explicado tres movimientos, se presenta en la siguiente página y el código respectivo es el siguiente:

```
proc hanoir {n o d a} {
  global l
  if {> $n 0} {
    set r [hanoir [- $n 1] $o $a $d]
    lappend $l [join [list $o "->" $d]]
    set r [hanoir [- $n 1] $a $d $o]
  }
}
```



Donde, sea empleado la variable global “l” para guardar los pasos que resuelven el juego. Dicha variable debe ser inicializada en una lista vacía:

```
set l {}
```

Inicializando “l” en una lista vacía, haciendo correr el módulo para n=3 y recuperando el valor de “l” se obtiene:

```
hecl> hanoir 3 "A" "C" "B"
hecl> set l
{A -> C} {A -> B} {C -> B} {A -> C} {B -> A} {B -> C} {A ->
C}
```

Que son los mismos movimientos deducidos manualmente en el ejemplo gráfico. Haciendo correr el módulo para n=8 se obtiene:

```
hecl> set l {}
hecl> hanoir 8 "A" "C" "B"
hecl> puts $l
{A -> B} {A -> C} {B -> C} {A -> B} {C -> A} {C -> B} {A ->
B} {A -> C} {B -> C} {B -> A} {C -> A} {B -> C} {A -> B} {A
-> C} {B -> C} {A -> B} {C -> A} {C -> B} {A -> B} {C -> A}
{B -> C} {B -> A} {C -> A} {C -> B} {A -> B} {A -> C} {B ->
C} {A -> B} {C -> A} {C -> B} {A -> B} {A -> C} {B -> C} {B
-> A} {C -> A} {B -> C} {A -> B} {A -> C} {B -> C} {B -> A}
{C -> A} {C -> B} {A -> B} {C -> A} {B -> C} {B -> A} {C ->
A} {B -> C} {A -> B} {A -> C} {B -> C} {A -> B} {C -> A} {C
-> B} {A -> B} {A -> C} {B -> C} {B -> A} {C -> A} {B -> C}
{A -> B} {A -> C} {B -> C} {A -> B} {C -> A} {C -> B} {A ->
B} {C -> A} {B -> C} {B -> A} {C -> A} {C -> B} {A -> B} {A
-> C} {B -> C} {A -> B} {C -> A} {C -> B} {A -> B} {C -> A}
{B -> C} {B -> A} {C -> A} {B -> C} {A -> B} {A -> C} {B ->
C} {B -> A} {C -> A} {C -> B} {A -> B} {C -> A} {B -> C} {B
-> A} {C -> A} {C -> B} {A -> B} {A -> C} {B -> C} {B -> A}
{C -> A} {C -> B} {A -> B} {A -> C} {B -> C} {B -> A} {C ->
A} {C -> B} {A -> B} {A -> C} {B -> C} {B -> A} {C -> A} {C
-> B} {A -> B} {C -> A} {B -> C} {B -> A} {C -> A} {C -> B}
{A -> B} {A -> C} {B -> C} {A -> B} {C -> A} {C -> B} {A ->
B} {A -> C} {B -> C} {B -> A} {C -> A} {B -> C} {A -> B} {A
-> C} {B -> C} {A -> B} {C -> A} {C -> B} {A -> B} {C -> A}
{B -> C} {B -> A} {C -> A} {C -> B} {A -> B} {A -> C} {B ->
C} {A -> B} {C -> A} {C -> B} {A -> B} {A -> C} {B -> C} {B
-> A} {C -> A} {B -> C} {A -> B} {A -> C} {B -> C} {B -> A}
{C -> A} {C -> B} {A -> B} {C -> A} {B -> C} {B -> A} {C ->
A} {C -> B} {A -> B} {C -> A} {B -> C} {A -> B} {A -> C} {B
-> C} {A -> B} {C -> A} {C -> B} {A -> B} {A -> C} {B -> C}
{B -> A} {C -> A} {B -> C} {A -> B} {A -> C} {B -> C} {B ->
A} {C -> A} {C -> B} {A -> B} {C -> A} {B -> C} {B -> A} {C
-> A} {C -> B} {A -> B} {C -> A} {B -> C} {A -> B} {A -> C}
{B -> C} {A -> B} {C -> A} {C -> B} {A -> B} {A -> C} {B ->
C} {A -> B} {C -> A} {C -> B} {A -> B} {A -> C} {B -> C} {B
-> A} {C -> A} {B -> C} {A -> B} {A -> C} {B -> C} {B -> A}
{C -> A} {C -> B} {A -> B} {C -> A} {B -> C} {B -> A} {C ->
A} {C -> B} {A -> B} {C -> A} {B -> C} {A -> B} {A -> C} {B
-> B} {A -> B} {C -> A} {B -> C} {B -> A} {C -> A} {C -> B}
{A -> B} {A -> C} {B -> C} {B -> A} {C -> A} {C -> B} {A ->
B} {C -> A} {B -> C} {B -> A} {C -> A} {B -> C} {A -> B} {A
-> C} {B -> C} {A -> B} {C -> A} {C -> B} {A -> B} {C -> A}
{B -> C} {B -> A} {C -> A} {B -> C} {A -> B} {A -> C} {B ->
```

```
C> {A -> B} {C -> A} {C -> B} {A -> B} {C -> A} {B -> C} {B
-> A} {C -> A} {C -> B} {A -> B} {A -> C} {B -> C} {A -> B}
{C -> A} {C -> B} {A -> B} {A -> C} {B -> C} {B -> A} {C ->
A} {B -> C} {A -> B} {A -> C} {B -> C} {B -> A} {C -> A} {C
-> B} {A -> B} {C -> A} {B -> C} {B -> A} {C -> A} {B -> C}
{A -> B} {A -> C} {B -> C} {A -> B} {C -> A} {C -> B} {A ->
B} {A -> C} {B -> C} {B -> A} {C -> A} {B -> C} {A -> B} {A
-> C} {B -> C}
hec1> llen $1
255
```

Y como se puede ver en este caso se requieren 255 movimientos para resolver el juego, pues el número de movimientos, en este juego, aumenta geométricamente con el número de discos, así con 16 discos se requieren 65535 movimientos.

Para evitar la necesidad de inicializar y mostrar manualmente la variable global, así como para realizar las comprobaciones adicionales, es conveniente crear otro módulo desde el cual se llama al módulo recursivo (que es donde realmente se resuelve el problema). El código de dicho módulo es el siguiente:

```
proc hanoi {n o d a} {
   if {!= [% $n 1] 0} {throw "El número de discos debe ser entero"}
   if {< $n 1} {throw "El número de discos debe ser mayor a 0"}
   global l
   set l {}
   hanoir $n $o $d $a
   return $l
}
```

Que por supuesto sigue devolviendo los resultados correctos, pero donde además se generan los errores respectivos:

```
hec1> hanoi 3 "A" "C" "B"
{A -> C} {A -> B} {C -> B} {A -> C} {B -> A} {B -> C} {A ->
C}
hec1> hanoi 6 "A" "C" "B"
{A -> B} {A -> C} {B -> C} {A -> B} {C -> A} {C -> B} {A ->
B} {A -> C} {B -> C} {B -> A} {C -> A} {B -> C} {A -> B} {A
-> C} {B -> C} {A -> B} {C -> A} {C -> B} {A -> B} {C -> A}
{B -> C} {B -> A} {C -> A} {C -> B} {A -> B} {A -> C} {B ->
C} {A -> B} {C -> A} {C -> B} {A -> B} {A -> C} {B -> C} {B
-> A} {C -> A} {B -> C} {A -> B} {A -> C} {B -> C} {B -> A}
{C -> A} {C -> B} {A -> B} {C -> A} {B -> C} {A -> B} {C ->
A} {B -> C} {A -> B} {A -> C} {B -> C} {B -> A} {C -> A} {C
-> B} {A -> B} {A -> C} {B -> C} {B -> A} {C -> A} {B -> C}
{A -> B} {A -> C} {B -> C}
hec1> hanoi 3.4 "A" "C" "B"
{ERROR {El n·mero de discos debe ser entero}} {throw 1} {if
El n·mero de discos debe ser entero
hec1> hanoi -4 "A" "C" "B"
{ERROR {El n·mero de discos debe ser mayor a 0}} {throw 1} {
        at org.hec1.Interp.run(Unknown Source)
El n·mero de discos debe ser mayor a 0
```

5. Invertir los Dígitos de un Número Entero

Como quinto ejemplo elaboraremos un módulo para invertir los dígitos de un número entero positivo o negativo, así por ejemplo si el número es 12345, el resultado deberá ser 54321.

Este problema fue resuelto con la estructura “while”, no obstante, el planteamiento recursivo es diferente: Si conocemos la inversa de los últimos “n-1” dígitos del número, podemos obtener la inversa del número multiplicando el mismo por 10 y sumando al resultado el primer dígito.

Así por ejemplo para invertir el número 12345, la operación recursiva es:

$$inversa(12345) = inversa(2345)*10+1 = 54320+1 = 54321$$

Como la inversa de 0 es 0, esta puede ser la condición de finalización.

Para separar los últimos n-1 dígitos del primero se debe dividir el número entre 10 elevado al número de dígitos menos 1, siendo el cociente los últimos "n-1" dígitos y el residuo el primer dígito. Así por ejemplo en el caso anterior, el residuo y el cociente de dicha división son:

```
hecl> % 12345 10000
2345
hecl> / 12345 10000
1
```

Por lo tanto la solución recursiva es:

$$inversa(n)=inversa(cociente(n/10^{nd(n)-1}))*10+residuo(n/10^{nd(n)-1})$$

$$inversa(0) = 0$$

El inconveniente para programar directamente la fórmula recursiva es que no contamos con un comando que devuelva el divisor, es decir 10 elevado al número de dígitos menos 1. Por ello, nos vemos forzados a elaborar un módulo que encuentre primero dicho número.

El razonamiento para dicho módulo será también recursivo: Si se conoce el valor de 10 elevado al número de dígitos menos uno del número sin su último dígito, el resultado se calcula multiplicando dicho valor por 10. Por ejemplo si el número es 12345 y la función que calcula 10 elevado al número de dígitos menos 1 es "dn1", resulta:

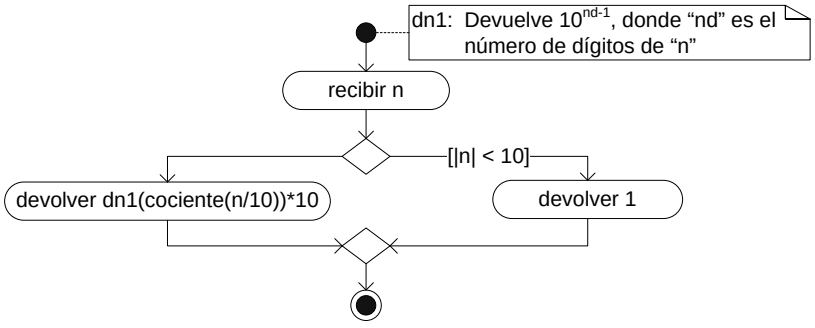
$$dn1(12345) = dn1(1234)*10 = 1000*10 = 10000$$

Si el número sólo tiene un dígito, entonces el resultado es 1, siendo esta la condición de finalización. Tomando en cuenta que el número menos su último dígito es simplemente el cociente de dicho número entre 10, la fórmula recursiva es:

$$dn1(n) = dn1(cociente(n/10))*10$$

$$dn1(n) = 1, \; si \; |n| < 10$$

El algoritmo para implementar esta expresión recursiva es:



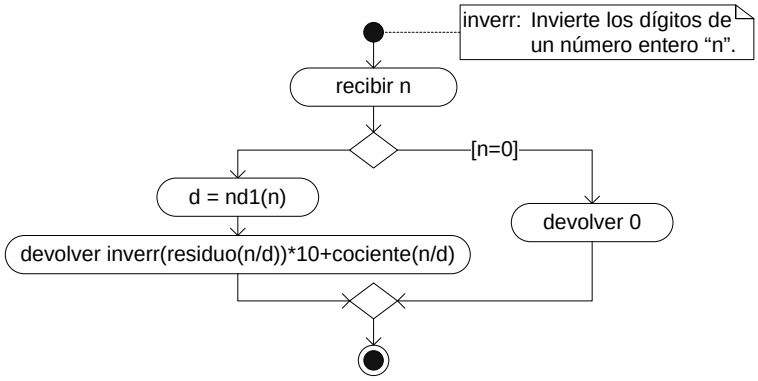
Y el código respectivo es:

```
proc dn1 n {
  if {< [abs $n] 10} {return 1} else {
    return [* [dn1 [/ $n 10]] 10]}
}
```

Haciendo correr el módulo con algunos valores de prueba se obtiene:

```
hecl> dn1 123
100
hecl> dn1 12345
10000
hecl> dn1 987675342
100000000
hecl> dn1 -47823434
10000000
```

Que como se puede verificar son los resultados correctos. Ahora que ya contamos con el módulo que nos devuelve 10 elevado al número de dígitos menos 1, podemos implementar la fórmula recursiva que invertir los dígitos de un número y cuya lógica es la siguiente:



Siendo el código respectivo:

```
proc inverr n {
  if {= $n 0} {return 0} else {
    set d [dn1 $n]
    + [* [inverr [% $n $d]] 10] [/ $n $d]
  }
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> inverr 12345
54321
hecl> inverr -8742394
-4932478
hecl> inverr 78323982
28932387
```

Que, como se puede observar, devuelve los resultados correctos.

Como ya se ha hecho en todos los casos anteriores, las verificaciones adicionales se las realiza en otro módulo desde el cual se llama luego al módulo recursivo:

```
proc inver n {
  if {!= [% $n 1] 0} {throw "El número debe ser entero"}
  inverr $n
}
```

Con este módulo se obtienen por supuesto los mismos resultados, pero además se genera un error cuando se manda un número real en lugar de uno entero:

```
hecl> inver 12345
54321
```

```
hecl> inver -482839423
-324938284
hecl> inver 45.53
<ERROR <El n·mero debe ser entero>> <throw 1> <if 2> <inver
        at org.hecl.Interp.run<Unknown Source>
El n·mero debe ser entero
```

8.3. EJERCICIOS

1. Elabore el diagrama de actividades y el código de un módulo recursivo que devuelva el número de dígitos que tiene un número entero (positivo o negativo) dado.

2. Tomando en cuenta que el capital acumulado en un banco en un determinadu número de periodos (por intereses recibidos en cada periodo) puede ser calculado si se conoce el capital acumulado hasta el periodo anterior. Elabore el diagrama de actividades y el código de un módulo recursivo que permita calcular dicho monto.

3. Elabore el diagrama de actividades y el código de un módulo recursivo que calcule la sumatoria de los números comprendidos entre 1 y el número entero recibido: $\sum i$ {i=1,2,3,4,...,n).

4. Elabore el diagrama de actividades y el código de un módulo recursivo que calcule la sumatoria de los primeros “n” números impares positivos.

5. Elabore el diagrama de actividades y el código de un módulo recursivo que calcule el común divisor entre dos números (A y B), mediante el algoritmo de Euclides (el módulo no debe volver a calcular valores calculados previamente):

$$\begin{aligned} &MCD(A,A) = A \\ &MCD(A,B) = MCD(B,A) \qquad si \quad A > B \\ &MCD(A,B) = MCD(A,B-A) \qquad si \quad A < B \end{aligned}$$

6. Elabore el diagrama de actividades y el código de un módulo recursivo que calcule la función de Ackerman para cualquier par de números enteros no negativos (el módulo no debe volver a calcular valores calculados previamente):

$$a(p,q) = \begin{cases} q+1 & si \quad p = 0 \\ a(p-1,1) & si \quad p <> 0 \quad y \quad q = 0 \\ a(p-1,a(p,q-1)) & si \quad p <> 0 \quad y \quad q <> 0 \end{cases}$$

7. Elabore el diagrama de actividades y el código de un módulo recursivo que calcule el Chebyshev enésimo ("n") de un número real "x": $C_n(x) = 2xC_{n-1}(x)-C_{n-2}(x)$; $C_0(x)=1$; $C_1(x)=x$). En el módulo no se deben volver a calcular valores calculados previamente.

8. Elabore el diagrama de actividades y el código de un módulo recursivo que calcule, la función "J" de Bessel de primera especie y orden "n", empleando la fórmula recursiva:

   $J_n(x)=(2n-2)J_{n-1}(x)/x-J_{n-2}(x)$;

   $J_0(x)=1-x^2/2^2+x^4/(2^2 4^2)-x^6/(2^2 4^2 6^2)+x^8/(2^2 4^2 6^2 8^2)-$
   $...+x^{24}/(2^2 4^2 6^2 8^2 10^2 12^2 16^2 18^2\ \ 20^2 22^2 24^2)$

$$J_1(x)=x/2-x^3/(2^2*4)+x^5/(2^2*4^2*6)-x^7/(2^2*4^2*6^2*8)+x^9/(2^2 4^2 6^2 8^2 10)-$$
$$...+x^{25}/(2^2 4^2\ 6^2 8^2 10^2 12^2 16^2 18^2 20^2 22^2 24^2 26).$$

9. ORDENACIÓN 1

Como una aplicación de las listas (conocidas también como vectores y matrices en otros lenguajes), en este tema estudiaremos tres métodos de ordenación. El propósito es que al concluir el tema estén capacitados para ordenar listas de elementos.

Ordenar es arreglar los elementos de una lista (o vector) de acuerdo a un determinado criterio. Por ejemplo para ordenar ascendentemente números reales, el criterio es que todos los elementos anteriores a un elemento dado sean menores o iguales a dicho elemento y que los elementos posteriores sean mayores o iguales al mismo. Si el objetivo es ordenar descendentemente los elementos de una vector con nombres, el criterio sería que todos los elementos anteriores a un elemento dado sean alfabéticamente, mayores o iguales a dicho elemento y que todos los elementos posteriores sean menores o iguales al mismo.

En la mayoría de los casos prácticos los elementos de un vector son ordenados para que sus elementos estén en orden ascendente o descendente. Es por ello que los métodos que estudiaremos en este capítulo nos permitirán ordenar los vectores en una de estas dos formas.

En general los métodos de ordenación suelen clasificarse en dos grupos: a) Los *métodos directos*, que ordenan comparando elementos consecutivos y b) los *métodos indirectos*, que ordenan comparando elementos que están separados entre sí por un cierto número de elementos.

En este tema estudiaremos tres métodos directos: *Burbuja*, *Selección* e *Inserción*. Los métodos del segundo grupo serán estudiados en el siguiente tema.

Con listas pequeñas (de hasta unos 1000 elementos) no existen diferencia importantes entre los distintos métodos, pero a medida que incrementa el número de elementos la diferencia entre el tiempo consumido por los métodos directos e indirectos también incrementa, hasta que con listas grandes (con cientos de miles o millones de elementos) las diferencias son tan grandes que los métodos directos pueden requerir días y hasta semanas para ordenar una lista mientras que los métodos indirectos pueden ordenarlas en cuestión de minutos.

En todos los métodos se explicará el procedimiento que se sigue para ordenar los elementos de manera ascendente pues para el orden inverso sólo es necesario cambiar la condición o pregunta (invirtiéndola).

9.1. MÉTODO DE BURBUJA

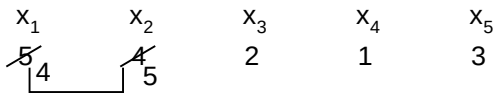
En este método se lleva el mayor valor la lista a la última posición. Para ello se comparan pares consecutivos de elementos (comenzando con los dos primeros) y se realiza un intercambio si el primer elemento es mayor que el segundo.

Una vez que se aplica el anterior procedimiento el mayor valor queda en la última posición, por lo tanto el último elemento queda ordenado. Entonces se aplica el mismo procedimiento con los “n-1” elementos restantes y así sucesivamente hasta que finalmente queda un solo elemento.

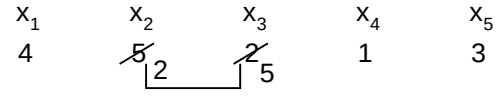
Veamos por ejemplo como se ordena el siguiente vector aplicando el método:

x_1	x_2	x_3	x_4	x_5
5	4	2	1	3

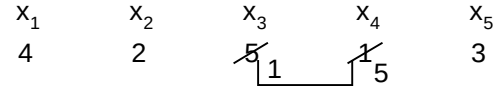
Comenzamos comparando los dos primeros valores (5 con 4) y puesto que el primero es mayor que el del segundo, se intercambian:



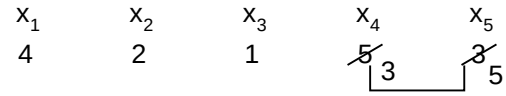
Ahora comparamos el segundo con el tercer elemento (5 con 2) y como el primero es mayor al segundo se intercambian:



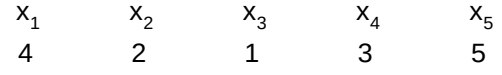
Continuamos con el tercer y cuarto elemento (5 con 1) y puesto que una vez más el primero es mayor que el segundo se vuelva a intercambiar:



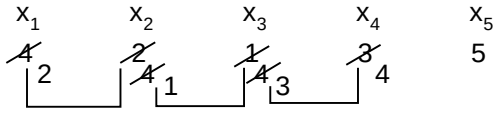
Finalmente comparamos el cuarto y quinto elemento (5 con 3) y se realiza otro intercambio:



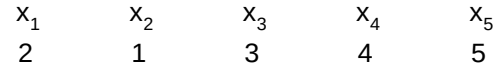
Con ello terminamos el procedimiento y como se puede observar, el mayor valor queda en la última posición:



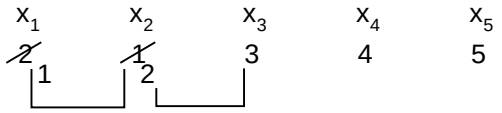
Ahora repetimos el mismo procedimiento pero sin tomar en cuenta el último elemento *(x_5=5)*, pues el ya está ordenado:



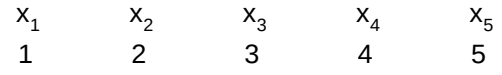
Con lo que ahora queda ordenado el penúltimo elemento:



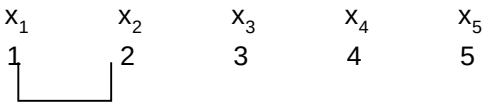
Repetimos ahora el proceso pero sin tomar en cuenta los dos últimos elementos (que ya están ordenados):



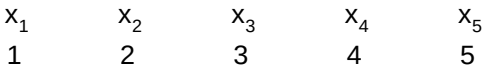
Con lo que el tercer elemento queda ordenado:



Finalmente comparamos el primer y segundo elementos:



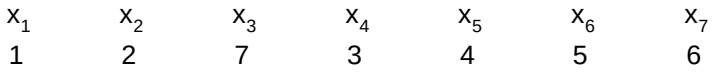
Con lo que el segundo elemento queda ordenado:



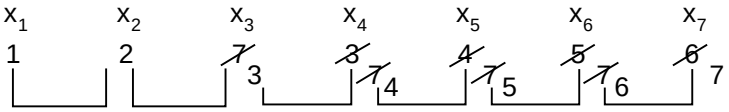
Y como solo resta un elemento, el proceso concluye y la lista queda ordenada.

Al aplicar el método de burbuja puede suceder que la lista quede ordenada antes de repetir el proceso *n-1* veces (inclusive la lista puede estar ordenada desde un principio). En el método de burbuja se sabe que la lista está ordenada cuando al aplicar el proceso no se produce ningún intercambio.

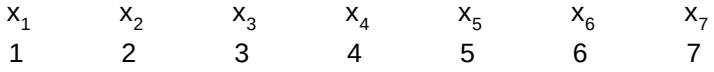
Por ejemplo si la lista a ordenar es la siguiente:



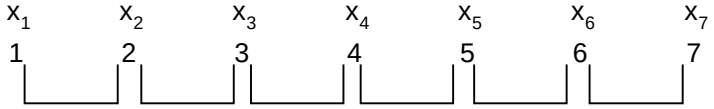
Aplicando el procedimiento de burbuja una vez se tiene:



Con lo que el vector queda ordenado:



Si aplicamos el procedimiento de burbuja una vez más:



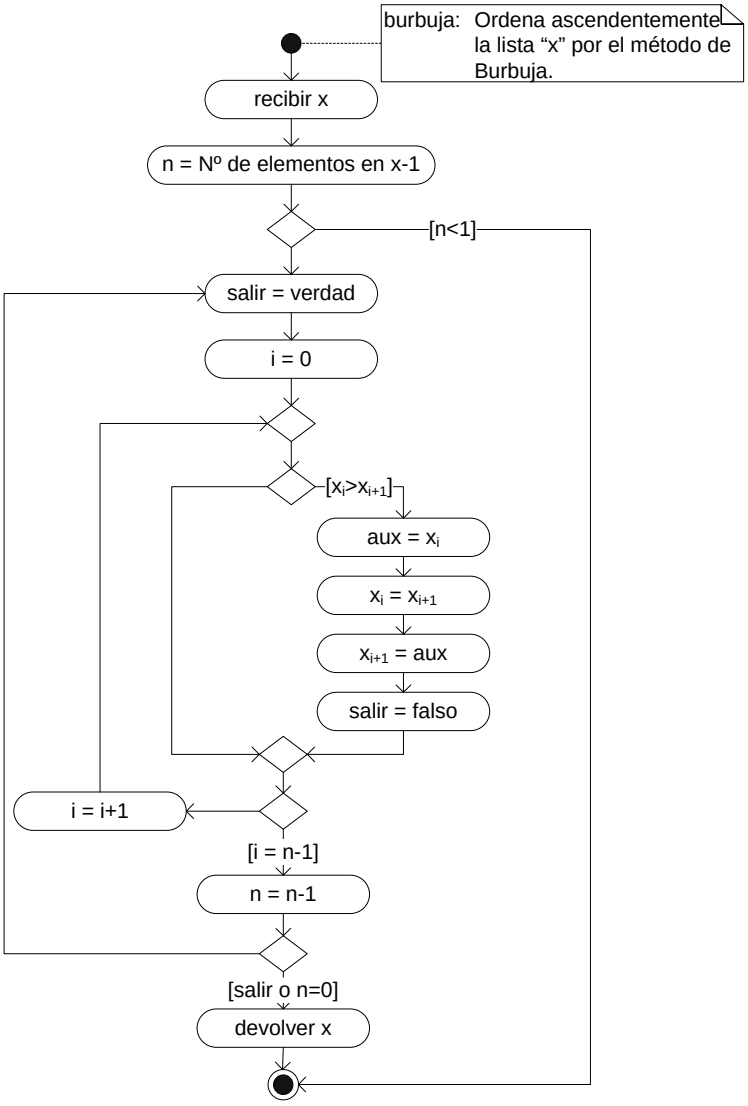
No se produce ningún intercambio, lo que nos indica que la lista ya está ordenada. En consecuencia si en una iteración del método no se realiza ningún intercambio es porque la lista ya está ordenada.

Todos los métodos que se estudian en este tema pueden ser empleados para ordenar listas con cualquier tipo de datos, sin embargo, los algoritmos serán desarrollados tomando en cuenta datos simples (principalmente números).

9.1.1. Algoritmo y código

El algoritmo del método de Burbuja, donde se sigue el proceso descrito en el anterior acápite, se muestra en la siguiente página y el código respectivo es el siguiente:

```
proc burbuja x {
    set n [- [llen $x] 1]
    if {< $n 1} {return $x}
    while true {
```




```
    set salir true
    for {set i 0} {<= $i [- $n 1]} {incr $i} {
      if {> [lindex $x $i][lindex $x [+ $i 1]]} {
        set aux [lindex $x $i]
        lset $x $i [lindex $x [+ $i 1]]
        lset $x [+ $i 1] $aux
        set salir false
      }
    }
    incr $i -1
    incr $n -1
    if {or [$salir] [= $n 0]} return
  }
}
```

Haciendo correr el programa con algunos valores de prueba se obtiene:

```
hecl> set l {7 9 2 3 4 1 8 3 2 4 10 13 6}
7 9 2 3 4 1 8 3 2 4 10 13 6
hecl> burbuja $l
1 2 2 3 3 4 4 6 7 8 9 10 13
hecl> set l { 45 -23 12 11 8 6 5 4 20 21 -32 -10 -5 1 9 }
 45 -23 12 11 8 6 5 4 20 21 -32 -10 -5 1 9
hecl> burbuja $l
-32 -23 -10 -5 1 4 5 6 8 9 11 12 20 21 45
```

Sin embargo, la verdadera prueba de la eficacia de estos métodos se la obtiene cuando se hacen correr los mismos no con unos cuantos términos, como en los anteriores ejemplos, sino con miles, cientos de miles o millones de términos. Por supuesto introducir datos manualmente para cientos, miles, cientos de miles y peor aún millones de elementos, consume demasiado tiempo. Por esa razón es conveniente contar con algún módulo que genere dichos valores.

Un ejemplo de dichos módulos, que genera números enteros aleatorios comprendidos entre dos límites, es el siguiente:

```
proc generarenteros {ne li ls} {
   if {< $ne 1} {throw "El N° de elementos a generar debe ser > 0"}
   set l {}
   set d [- $ls $li]
   for {set i 1} {<= $i $ne} {incr $i} {
     lappend $l [int [+ [* [random] $d] $li]]
   }
   return $l
}
```

Así por ejemplo para generar una lista con 100 elementos comprendidos entre 1 y 100, guardándola en la variable “l”, escribimos:

```
hecl> set l [generarenteros 100 1 100]
85 18 91 55 60 95 95 19 39 11 46 46 97 55 75 7 56 41 88 77 8
5 18 56 59 16 72 63 41 59 77 92 68 41 95 96 38 9 54 38 98 87
 93 24 87 78 2 40 18 80 72 81 68 32 7 72 72 40 60 65 14 44 2
4 4 14 27 98 86 14 47 39 58 84 97 83 80 29 64 79 14 23 52 31
 46 95 89 6 61 43 17 63 26 85 92 54 13 95 87 52 54 49
```

Ahora podemos ordenar esta lista con “burbuja”, pero para comparar la eficiencia de dos o más métodos es conveniente controlar el tiempo que requieren para ordenar una lista, algo que en “hecl” se logra con la instrucción “time”:

time *{instrucciones}*

Que devuelve el tiempo, en milisegundos, que toma ejecutar las instrucciones que se le mandan como parámetro.

```
hecl> time {burbuja $l}
125
hecl> puts $l
2 4 6 7 7 9 11 13 14 14 14 14 16 17 18 18 18 19 23 24 24 26
27 29 31 32 38 38 39 39 40 40 41 41 41 43 44 46 46 46 47 49
52 52 54 54 54 55 55 56 56 58 59 59 60 60 61 63 63 64 65 68
68 72 72 72 72 75 77 77 78 79 80 80 81 83 84 85 85 85 86 87
87 87 88 89 91 92 92 93 95 95 95 95 95 96 97 97 98 98
```

Como se puede ver, el método de burbuja requiere (en la computadora en la que se ha hecho correr el ejemplo) 135 milisegundos para ordenar la lista. Si ahora hacemos la prueba con una lista de 500 elementos obtenemos (para ahorrar espacio, sólo se muestran las dos primeras filas de elementos):

```
hecl> set l [generarenteros 500 1 500]
478 109 488 26 251 436 36 309 66 167 311 399 302 470 57 22 4
60 90 94 157 455 151 471 106 478 141 354 375 12 58 76 118 84
```

```
hecl> time {burbuja $l}
2141
hecl> puts $l
4 4 6 7 7 8 11 12 13 13 14 16 16 19 21 21 21 22 22 22 22 23
23 24 24 26 26 27 29 29 30 30 30 32 33 33 33 36 36 38 40 44
```

Ahora "burbuja" requiere 2141 milisegundos, es decir ¡casi 16 veces más del tiempo requerido para ordenar una lista con 100 elementos!, no cinco veces más como podría esperarse. Esto es algo característico de los métodos directos: **el tiempo requerido incrementa geométricamente con el número de elementos a ordenar** y esa es la razón por la cual se emplean sólo para ordenar listas pequeñas.

Si ahora generamos y ordenamos 2000 elementos (donde una vez más sólo se muestran las dos primeras filas de elementos), obtenemos:

```
hecl> set l [generarenteros 2000 1 2000]
1460 496 511 1445 736 729 428 1276 1817 13 1592 1750 386 146
2 1686 1191 339 498 1455 635 475 142 265 321 1909 1118 855 1
```

```
hecl> time {burbuja $l}
35375
hecl> puts $l
2 2 2 3 4 8 10 11 11 11 11 13 13 13 14 14 14 18 18 18 19 19
21 22 23 24 24 24 26 31 32 32 33 33 34 34 35 37 38 39 40 41
```

Es decir algo más de 35 segundos: ¡283 veces el tiempo requerido para ordenar 100 elementos! no 20 veces como se esperaría si el incremento fuera aritmético. Debido a este incremento geométrico, el método de burbuja sólo resulta de utilidad práctica para ordenar unos cientos de elementos (una lista con unos 10000 elementos fácilmente puede requerir más de 30 minutos para ser ordenada por este método).

9.2. MÉTODO DE SELECCIÓN

En este método se selecciona el mayor valor de la lista y se intercambia con el último elemento, de esta manera (al igual que sucede con el método de *Burbuja*), el mayor valor queda siempre en la última posición.

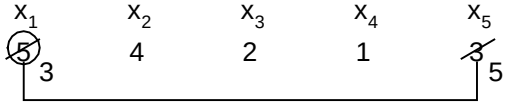
Como el último elemento ya está ordenado el procedimiento se repite con los elementos restantes (los n-1 elementos restantes) y se continúa así (n-1 veces) hasta que sólo queda un elemento sin ordenar en la lista.

Este método suele ser un tanto más eficiente que el método de Burbuja, pues requiere menos intercambios, sin embargo, en este método no existe ningún indicador que nos permita determinar si la lista ya está ordenada, por lo que siempre se repite *n-1* veces inclusive si la lista ya está ordenada desde un principio.

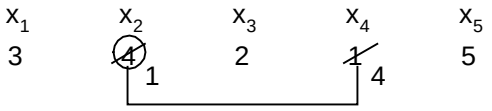
Para comprender mejor el método ordenemos la siguiente lista aplicando el método.

x_1	x_2	x_3	x_4	x_5
5	4	2	1	3

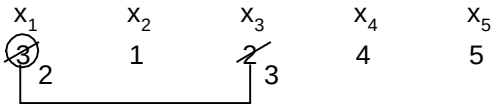
Seleccionamos el mayor de valor de la lista (el número 5) e intercambiamos el mismo con el último elemento:



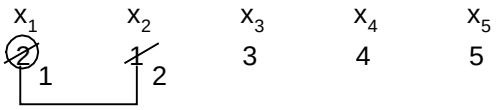
Ahora repetimos el procedimiento sin tomar en cuenta el último elemento (pues ya está ordenado):



Repetimos el procedimiento sin tomar en cuenta ahora los dos últimos elementos (que ya están ordenados):



Finalmente repetimos una vez más el procedimiento sin tomar en cuenta los tres últimos elementos (que ya están ordenados):



Con lo que el vector queda ordenado:

x_1	x_2	x_3	x_4	x_5
1	2	3	4	5

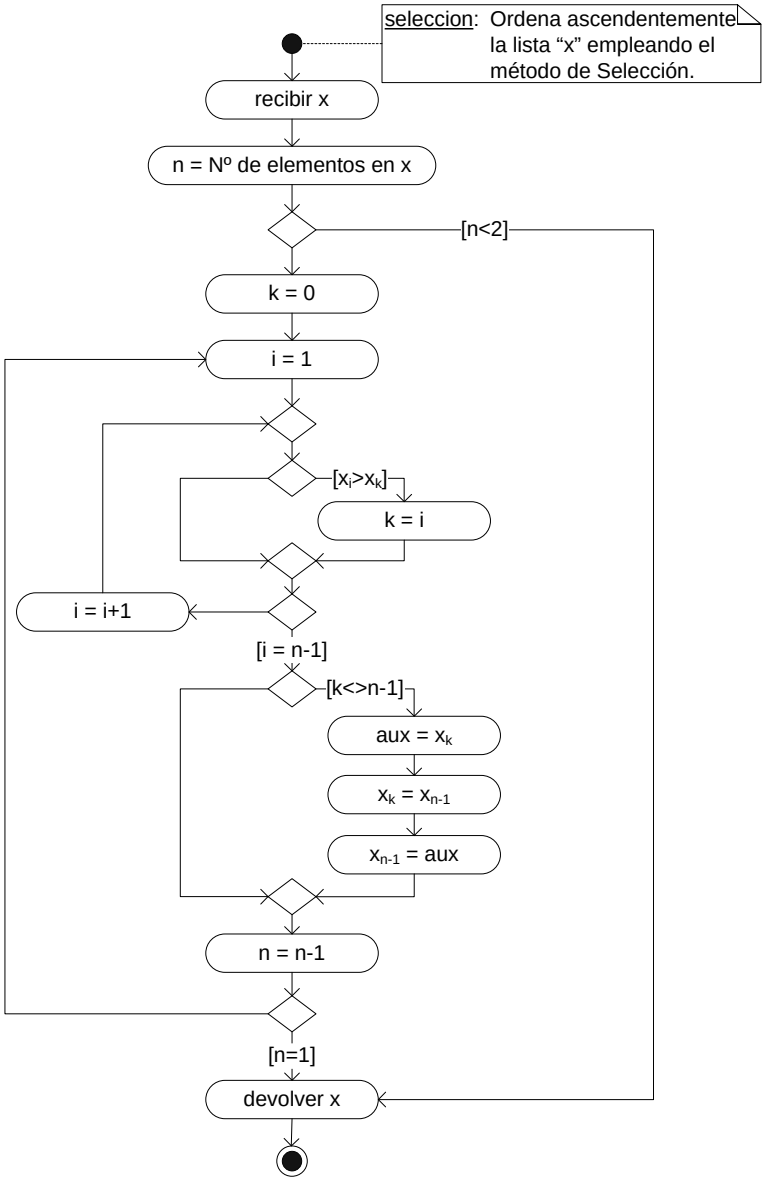
9.2.1. Algoritmo y código

Como se ha podido apreciar en el ejemplo, el algoritmo del método de selección es muy simple: se ubica el mayor elemento de la lista y se intercambia el mismo con el último valor no ordenado, repitiendo el proceso n-1 veces.

Dicho algoritmo en forma de diagrama de actividades se presenta en la siguiente página y el código respectivo es el siguiente:

```
proc seleccion x {
  set n [llen $x]
  if {< $n 2} return
  for {} {!= $n 1} {incr $n -1} {
    for {set k 0; set i 1} {<= $i [- $n 1]} {incr $i} {
      if {> [lindex $x $i][lindex $x $k]} {set k [copy $i]}
    }
    if {!= $k [- $n 1]} {
      set aux [lindex $x $k]
      lset $x $k [lindex $x [- $n 1]]
      lset $x [- $n 1] $aux
    }
  }
}
```

Al igual que en el método de burbuja, el módulo no devuelve nada (la instrucción “return” no tiene parámetro). Se procede así porque como ya se dijo: con listas “hecl” trabaja con valores por referencia (es decir con direcciones de memoria), por lo tanto, cualquier cambio que se hace a la lista, en el módulo, es un cambio a la matriz original, por consiguiente, la matriz original queda ordenada cuando el módulo termina. Por la misma razón el valor del contador “i” se pasa a la variable “k” con “copy”, instrucción que permite copiar realmente el valor de la variable y no sólo su dirección.



Haciendo correr el módulo con 100 elementos, resulta:

```
hecl> set l [generarenteros 100 1 100]
47 61 73 93 67 16 90 35 61 75 66 37 33 45 92 53 72 29 55 6 6
7 20 81 5 38 83 4 11 69 27 85 66 38 27 13 96 6 27 78 80 28 8
4 68 47 86 66 8 7 27 73 22 25 14 19 5 16 58 96 22 60 13 70 5
5 56 25 32 50 88 70 25 38 3 90 66 17 8 16 70 56 58 88 74 19
99 33 42 21 94 88 51 72 33 1 87 65 30 76 39 57 49
```

```
hecl> time {seleccion $l}
47
hecl> puts $l
1 3 4 5 5 6 6 7 8 8 11 13 13 14 16 16 16 17 19 19 20 21 22 2
2 25 25 25 27 27 27 27 28 29 30 32 33 33 33 35 37 38 38 38 3
9 42 45 47 47 49 50 51 53 55 55 56 56 57 58 58 60 61 61 65 6
6 66 66 66 67 67 68 69 70 70 70 72 72 73 73 74 75 76 78 80 8
1 83 84 85 86 87 88 88 88 90 90 92 93 94 96 96 99
```

Es decir se requieren 47 milisegundos para ordenar 100 elementos. Con 500 elementos (mostrando sólo la primera fila de elementos) se obtiene:

```
hecl> set l [generarenteros 500 1 500]
341 2 491 406 282 99 283 17 223 406 95 267 283 218 350 457 2
```

```
hecl> time {seleccion $l}
906
hecl> puts $l
1 1 2 3 6 6 7 7 7 7 7 7 9 9 9 9 10 11 11 11 13 15 15 16 17 1
```

Es decir se requieren algo más de 19 veces el tiempo requerido para ordenar 100 elementos (no 5 veces más como podría esperarse).

Con 2000 elementos resulta:

```
hecl> set l [generarenteros 2000 1 2000]
1622 90 1973 684 209 478 1061 163 566 1532 1706 770 1442 175
```

```
hecl> time {seleccion $l}
14063
hecl> puts $l
1 2 3 7 9 10 11 12 12 13 13 14 15 15 16 17 19 20 23 24 24 24
```

Es decir casi 300 veces el tiempo requerido para ordenar 100 elementos (no 20 como podría esperarse). En casi todos los casos este método requiere menos de la mitad del tiempo que requiere el método de Burbuja, aunque como se puede ver el incremento del tiempo con el número de elementos sigue siendo geométrico.

9.3. MÉTODO DE INSERCIÓN

En este método se inserta un elemento en su posición correcta con relación a los elementos que se encuentran antes del mismo. Cuando la lista está totalmente desordenada, el método comienza con el segundo elemento y continúa insertando elementos, en su posición correcta con relación a los anteriores, hasta llegar al último elemento.

En la práctica este método se emplea principalmente para insertar nuevos elementos en listas ya ordenadas pero no para ordenar todo el vector, pues como veremos luego para ello existen métodos más eficientes, no obstante, en este tema implementaremos el método de manera que permita ordenar listas completas.

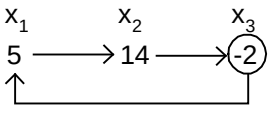
Para comprender mejor el método ordenemos el siguiente vector aplicando el procedimiento:

x_1	x_2	x_3	x_4	x_5
5	14	-2	10	3

Comenzamos con el segundo elemento (que tiene el valor 14) y como es mayor al primero se encuentra ya en su posición correcta con relación al elemento anterior, por lo que es insertado directamente en la segunda posición:

x_1	x_2	x_3	x_4	x_5
5	(14)	-2	10	3

Ahora pasamos el tercer elemento (que tiene el valor -2) y como es menor a los dos elementos anteriores, es insertado en la primera posición, desplazando los elementos anteriores una posición hacia la derecha:



Con ello los tres primeros elementos quedan ordenados ascendentemente:

x_1	x_2	x_3	x_4	x_5
-2	5	14	10	3

Ahora pasamos al cuarto elemento (que tiene el valor 10) y como este elemento es menor al tercero, pero mayor al segundo, debe ser insertado en la tercera posición:

x_1	x_2	x_3	x_4	x_5
-2	5	14 →	(10)	3

Quedando los cuatro primeros elementos ordenados:

x_1	x_2	x_3	x_4	x_5
-2	5	10	14	3

Finalmente pasamos al quinto elemento (que tiene el valor 3) y como este elemento es menor al segundo, pero mayor al primero, debe ser insertado en la segunda posición:

x_1	x_2	x_3	x_4	x_5
-2	5 →	10 →	14 →	(3)

Quedando así los 5 elementos ordenados:

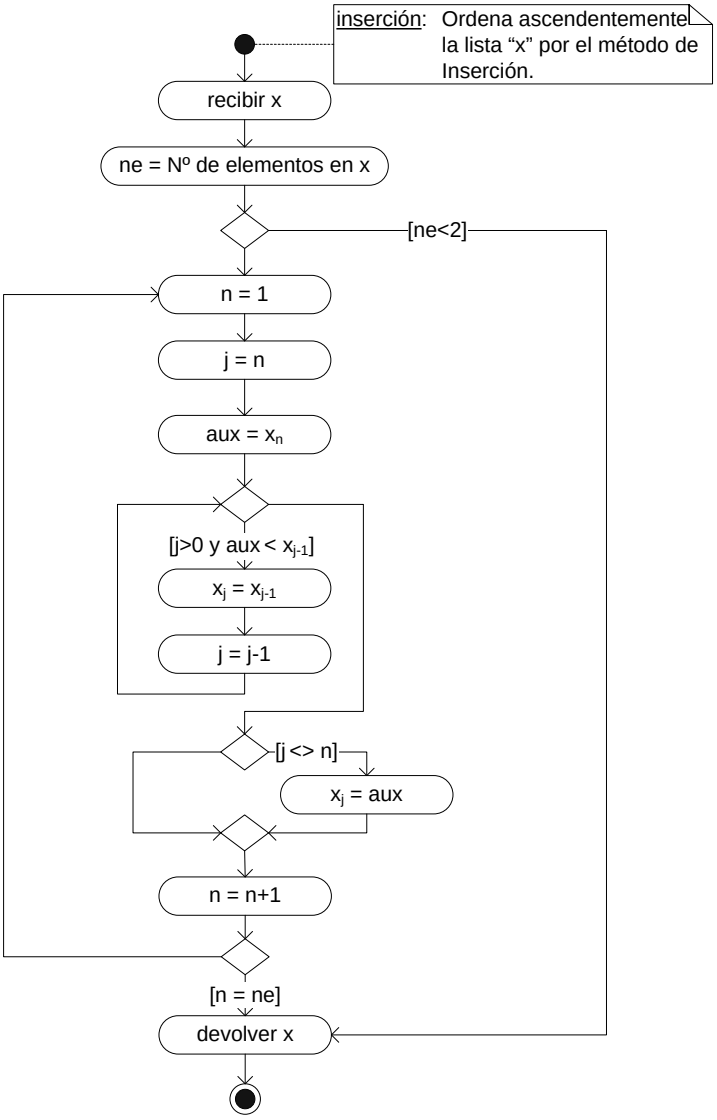
x_1	x_2	x_3	x_4	x_5
-2	3	5	10	14

9.3.1. Algoritmo y código

El algoritmo del método de inserción, en forma de diagrama de actividades, se presenta en la siguiente página y el código elaborado en base al mismo es el siguiente:

```
proc insercion x {
  set ne [llen $x]
  if {< $ne 2} return
  set n 1
  while {< $n $ne} {
    set j [copy $n]
    set aux [lindex $x $n]
    while {and [> $j 0][< $aux [lindex $x [- $j 1]]]} {
      lset $x $j [lindex $x [- $j 1]]
      incr $j -1
    }
    if {!= $j $n} {lset $x $j $aux}
    incr $n
  }
}
```

Una vez más, el módulo no devuelve en realidad ningún resultado, pues con listas se trabaja con valores por referencia. Por la misma razón se emplea la instrucción “copy”, para pasar el valor de “n” a “j”.



Haciendo correr el programa con 100 elementos se obtiene:

```
hecl> set l [generarenteros 100 1 100]
81 39 54 45 60 42 8 69 75 23 4 64 79 40 93 86 46 88 64 38 66
 14 77 29 39 11 81 76 62 41 14 94 6 68 92 92 44 47 25 5 54 9
1 1 35 73 58 94 27 33 31 46 92 99 48 40 44 61 18 73 10 38 93
 52 12 4 36 13 79 38 52 60 87 55 44 38 11 13 6 73 29 84 87 6
5 72 5 49 91 5 43 13 99 67 94 26 91 47 60 71 56 4
hecl> time {insercion $l}
47
hecl> puts $l
1 4 4 4 5 5 5 6 6 8 10 11 11 12 13 13 13 14 14 18 23 25 26 2
7 29 29 31 33 35 36 38 38 38 38 39 39 40 40 41 42 43 44 44 4
4 45 46 46 47 47 48 49 52 52 54 54 55 56 58 60 60 60 61 62 6
4 64 65 66 67 68 69 71 72 73 73 73 75 76 77 79 79 81 81 84 8
6 87 87 88 91 91 91 92 92 92 93 93 94 94 94 99 99
```

Por lo tanto “inserción” requiere 47 milisegundos para ordenar una lista con 100 elementos. Haciendo la prueba con 1000 elementos (pero mostrando sólo las dos primeras filas) se tiene:

```
hecl> set l [generarenteros 1000 1 1000]
850 130 407 519 549 603 963 368 168 295 842 111 522 543 689
219 1 148 55 85 804 150 414 9 643 948 675 408 808 882 599 96
```

```
hecl> time {insercion $l}
2156
hecl> puts $l
1 1 4 4 4 4 4 4 7 9 9 10 10 10 11 12 12 13 14 14 16 17 18 20
 21 22 26 30 30 30 32 37 38 38 40 41 42 42 43 44 46 47 47 50
```

Es decir que “inserción” requiere casi 46 veces el tiempo requerido para ordenar 100 elementos (no 10 como sería de esperar si la relación fuera aritmética). Sin embargo se trata de una progresión geométrica más favorable que la de los dos métodos anteriores.

Con 5000 elementos se tiene:

```
hecl> set l  [generarenteros 5000 1 5000]
2668 119 3950 2340 3248 977 1856 3889 1080 2741 4929 897 101
 795 2880 1263 1487 3576 2369 4698 100 901 888 202 2009 481
```

```
hecl> time {insercion $l}
53907
hecl> puts $l
1 1 1 2 2 3 7 8 10 11 13 13 13 14 23 24 24 25 26 26 27 28 28
 29 29 30 30 31 32 32 34 34 35 35 36 39 40 41 41 42 44 47 48
```

Es decir que se requiere casi 1147 veces el tiempo requerido para ordenar 100 elementos (no 50). Una vez más esta relación es más favorable que la de los anteriores métodos, por lo que podemos concluir que de los métodos estudiados hasta el momento, el método de inserción es el más eficiente.

9.4. EJERCICIOS

1. Elabore un módulo que ordene descendentemente una lista empleando el método de Burbuja.

2. Elabore un módulo recursivo, que ordene ascendentemente una lista empleando el método de Burbuja.

3. Elabore un módulo que genere “n” números reales aleatorios comprendidos entre dos límites dados.

4. Elabore un módulo que ordene descendentemente una lista empleando el método de Selección.

5. Elabore un módulo recursivo que ordene ascendentemente una lista empleando el método de Selección.

6. Elabore un módulo que ordene descendentemente listas empleando el método de inserción.

7. Elabore un módulo que inserte un nuevo elemento, en una lista ordenada ascendentemente, empleando el método de inserción.

10. ORDENACIÓN 2

Como ya se mencionó en el anterior tema, en el presente se estudiarán dos métodos indirectos (es decir aquellos en los que no se comparan valores sucesivos): Shell y Quick Sort.

10.1. MÉTODO SHELL

El método *Shell*, denominado así en honor a su creador *D. L. Shell*, es el primero de los métodos indirectos que estudiaremos en este tema.

En este método se comparan valores que están separados entre sí por un determinado número de elementos y al igual que en el método de burbuja si el primer valor es mayor que el segundo se intercambian.

Al número de elementos que separan los valores a comparar se conoce con el nombre de *salto* (o *paso*) e inicialmente es igual a la mitad (entera) del número de elementos existentes.

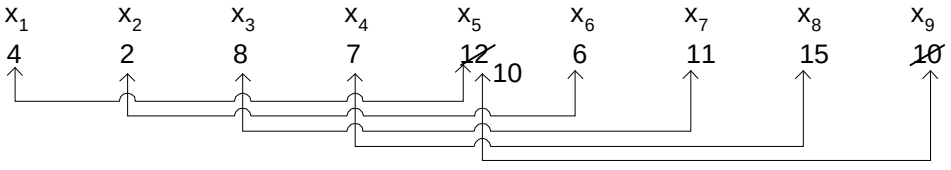
Si *k* es el paso o salto, en el proceso se comparan el primer elemento con el elemento que se encuentra *k+1* elementos más adelante, luego se compara el segundo elemento con el elemento que se encuentra *k+2* elementos más adelante y así sucesivamente hasta que se compara el último elemento de la lista, es decir se realiza un total de *k-p* comparaciones.

El anterior procedimiento se repite hasta que no ocurre ningún intercambio, entonces el *paso* o *salto* es reducido a la mitad y el procedimiento se vuelve a repetir (con el nuevo *paso*) hasta que una vez más no existen más intercambios. Cuando esto ocurre se vuelve a reducir el valor del *paso* a la mitad y se continúa de esta manera hasta que el *paso* se reduce a 1.

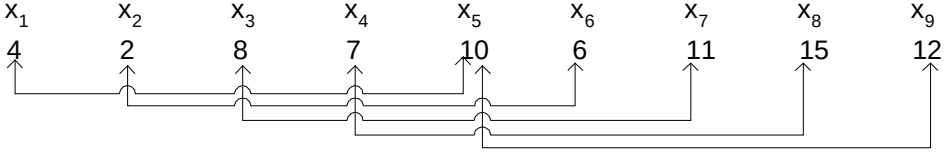
Para comprender mejor el procedimiento ordenaremos el siguiente vector siguiendo el método Shell:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
4	2	8	7	12	6	11	15	10

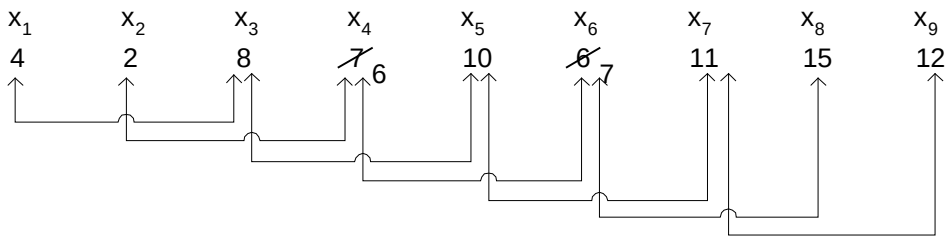
Como el número de elementos es 9, el paso inicial es 9/2=4 (división entera). Entonces comparamos el primer elemento con el quinto, el segundo con el sexto y así sucesivamente:



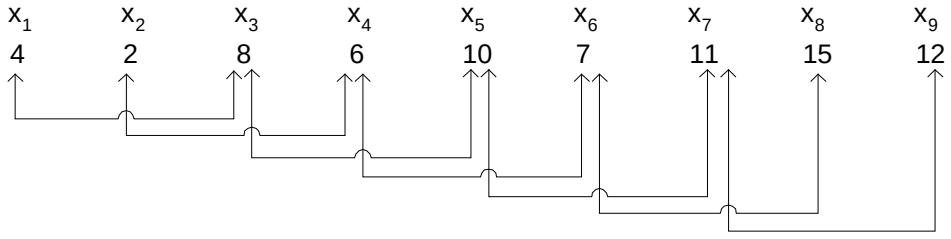
Y como se ha producido un intercambio repetimos el procedimiento:



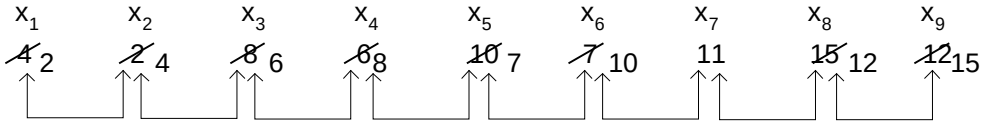
Como ahora no se produce ningún intercambio, reducimos el paso a la mitad de su valor. Por lo tanto el nuevo paso será: 4/2=2, entonces comparamos x_1 con x_3, x_2 con x_4 y así sucesivamente:



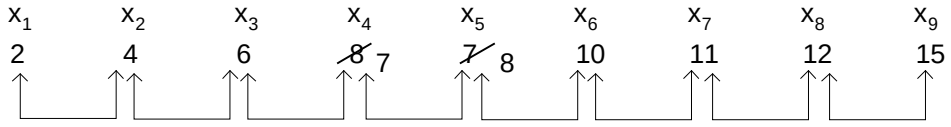
Como se ha producido un intercambio repetimos el proceso:



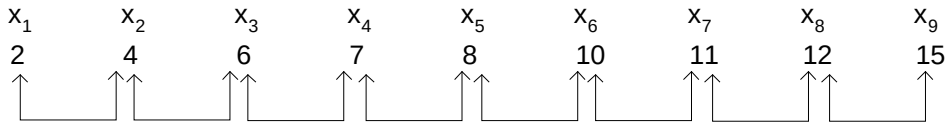
Y dado que no se produce ningún intercambio, reducimos el paso a la mitad: 2/2=1. Entonces comparamos elementos sucesivos: el primero con el segundo, el segundo con el tercero, etc.:



Como han existido intercambios, se repite el procedimiento:



Como se ha producido un intercambio se sigue repitiendo el procedimiento:

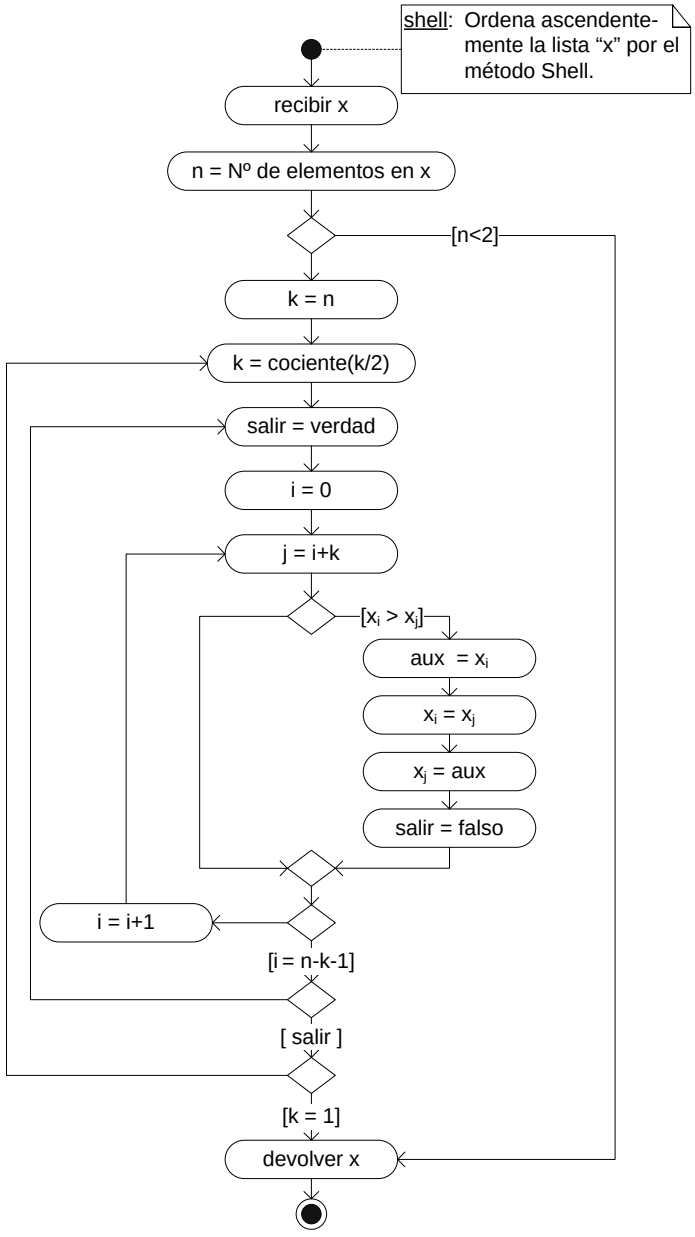


Puesto que ahora no se produce ningún intercambio y dado que el paso ya es 1, el proceso concluye y como se ve la lista queda ordenada.

Este método es más eficiente que los métodos estudiados hasta ahora porque requiere un menor número de intercambios. Así, inclusive con el vector de tan sólo 9 elementos del ejemplo, se requieren 7 intercambios, mientras que con *Burbuja* se habrían llevado a cabo por lo menos 9. Esta diferencia es más grande cuanto mayor es el número de elementos del vector.

10.1.1. Algoritmo y código

El algoritmo del método Shell, donde se sigue el procedimiento descrito en el acápite anterior, se presenta en el diagrama de actividades de la siguiente página:



Siendo el código respectivo el siguiente:

```
proc shell x {
  set n [llen $x]
  if {< $n 2} return
  set k $n
  while true {
    set k [/ $k 2]
    while true {
      set salir true
      for {set i 0} {<= $i [- $n $k 1]} {incr $i} {
        set j [+ $i $k]
        if {> [lindex $x $i][lindex $x $j]} {
```

```
            set aux [lindex $x $i]
            lset $x $i [lindex $x $j]
            lset $x $j $aux
            set salir false
          }
        }
        if $salir break
      }
      if {= $k 1} break
    }
  }
```

Haciendo correr el programa con 10 valores generados al azar se obtiene:

```
hecl> set l [generarenteros 10 1 10]
9 3 3 9 2 2 2 3 2 6
hecl> time {shell $l}
0
hecl> puts $l
2 2 2 2 3 3 3 6 9 9
```

Para comparar la eficiencia de este método con alguno de los del tema anterior, ordenaremos 2000 elementos con el método Shell y con el método de Burbuja:

```
hecl> set l [generarenteros 2000 1 2000]
1373 1305 1336 362 713 82 1967 1891 1124 1369 1447 498 1062
1014 625 384 1028 787 1174 672 1101 947 521 1866 1034 401 19
43 1934 1546 1941 1732 1857 1447 1382 1518 240 104 1 1879 56
```

```
hecl> set r [copy $l]
1373 1305 1336 362 713 82 1967 1891 1124 1369 1447 498 1062
1014 625 384 1028 787 1174 672 1101 947 521 1866 1034 401 19
43 1934 1546 1941 1732 1857 1447 1382 1518 240 104 1 1879 56
```

```
hecl> time {shell $l}
2500
hecl> time {burbuja $r}
33797
hecl> puts $l
1 3 4 5 6 8 9 9 9 9 9 10 12 13 13 13 14 14 15 16 16 20 23 24
 24 26 26 26 27 28 28 29 32 33 33 33 33 33 35 37 38 39 41 41
 43 44 44 44 45 46 47 47 48 49 49 50 51 51 51 54 54 55 56 57
```

Como se puede ver Shell demora 2.5 segundos para ordenar los 2000 elementos, mientras que burbuja demora más de 33 segundos para ordenar los mismos 2000 elementos, es decir requiere más de 13 veces el tiempo que requiere Shell.

Esta diferencia se va acentuando a medida que incrementa el tamaño de la lista, así para 10000 elementos:

```
hecl> set l [generarenteros 10000 1 10000]
2586 7391 1536 5817 9595 332 438 1678 4554 8650 9012 1292 92
77 1513 3336 1899 860 5543 5061 4310 5236 1702 2601 8499 794
9 2853 4740 2007 6668 5372 1451 78 7672 2154 5123 2495 328 4
```

```
hecl> time {shell $l}
19203
hecl> puts $l
1 1 2 2 6 6 7 10 10 11 11 13 16 16 17 17 17 18 18 18 19 20 2
0 20 20 21 21 22 24 24 24 25 26 27 27 30 31 33 33 35 35 36 3
6 37 38 39 40 41 42 42 43 46 47 49 50 50 50 51 53 53 53 55 5
```

Shell requiere un poco más de 19 segundos, mientras que burbuja demora ¡más de una hora! para lograr el mismo resultado.

10.2. MÉTODO QUICK-SORT (HOARE)

Este método fue propuesto por *C. R. Hoare* y es el método de ordenación más eficiente de todos los que se estudiarán.

Básicamente el método consiste en dividir el vector en dos: uno con elementos menores o iguales a un valor de referencia denominado pivote y otro con elementos mayores o iguales a dicho valor.

El anterior procedimiento se repite con cada uno de los dos vectores resultantes y luego con cada uno de los vectores resultantes de estos y así sucesivamente hasta que cada vector queda con un solo elemento.

El valor de referencia, el valor *pivote*, puede ser cualquiera de los elementos del vector, aunque frecuentemente se elige el elemento central, el primer elemento o el último elemento. Al finalizar la división, el *pivote* queda en uno de los dos vectores resultantes o al medio de los dos, en cuyo caso se encuentra en ya en la posición correcta.

Para comprender mejor el procedimiento se ordenará la siguiente lista empleando el método *Quick Sort*:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
15	21	30	7	20	18	10	14	28

Para este ejemplo, se elige como *pivote* el primer elemento, por lo tanto el *pivote* será 15 (tome en cuenta que el pivote es el valor del primer elemento, no el elemento en sí, porque en el proceso su valor puede ir cambiando). Ahora debemos dividir la lista en 2, uno con elementos mayores a 15 y otro con elementos menores a 15. Para ello se emplean dos contadores: uno que comienza en el índice más bajo del vector (para los datos del ejemplo 1) y otro en el índice más alto (para los datos del ejemplo 9):

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	
15	21	30	7	20	18	10	14	28	pivote =15
i=1								j=9	

Ahora se incrementa el valor del contador *i* mientras x_i sea menor al valor pivote. Como en este caso x_i es igual al pivote, no incrementa su valor. Se pasa entonces a disminuir el valor del contador *j*, en este caso mientras x_j sea mayor al pivote, por lo tanto el contador *j* disminuye hasta x_8, donde el valor (14) es menor al pivote:

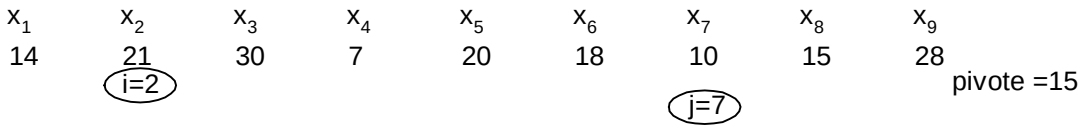
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	
15	21	30	7	20	18	10	14	28	pivote =15
i=1							j=8	j=9	

Entonces se intercambian x_i con x_j, (x_1 con x_8), luego se incrementa el contador *i* en uno y se disminuye el contador *j* en uno:

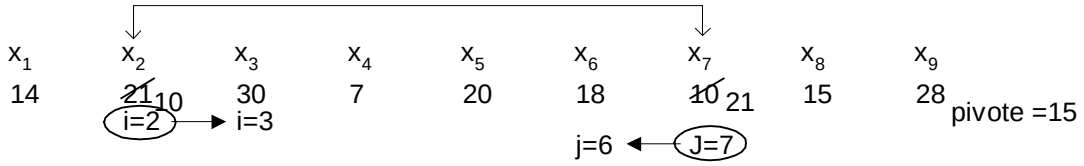
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	
~~15~~ 14	21	30	7	20	18	10	14 15	28	pivote =15
i=1 →	i=2					j=7 ←	j=8		

Ahora volvemos a repetir el proceso, es decir se incrementa el contador *i* mientras x_i sea menor al pivote y se disminuye *j* mientras x_j sea mayor al

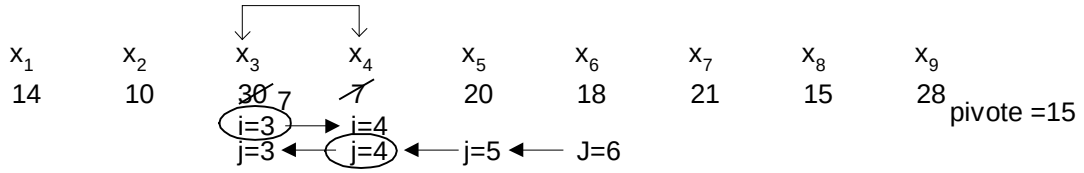
pivote. En este caso x_i ya es mayor al pivote y x_j menor, por lo tanto no incrementan ni se disminuyen sus valores:



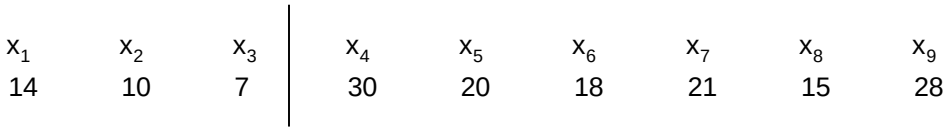
Ahora que los contadores han quedado fijos, intercambiamos x_i con x_j, luego se incrementa i en uno y se disminuye j en uno:



Ahora volvemos a repetir el procedimiento: incrementar i mientras x_i sea menor al pivote, disminuir j mientras x_j sea mayor al pivote, intercambiar variables, incrementar i y disminuir j:

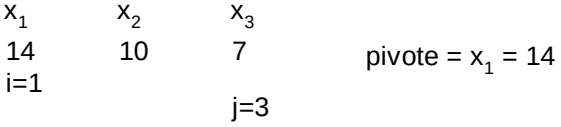


Como vemos en este caso el contador i queda con un valor mayor que el contador j. Cuando esto sucede el proceso concluye y la lista queda dividida en dos: la lista izquierda que va desde el primer elemento hasta j (es decir desde 1 hasta 3) y la derecha que va desde i hasta el último elemento (es decir desde 4 hasta 9):

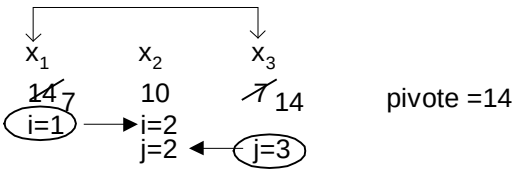


De esta manera se consigue dividir el vector en dos: el izquierdo con los elementos menores al pivote y el derecho con los elementos mayores o iguales al pivote.

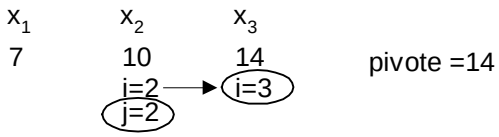
Ahora se vuelve a repetir el mismo procedimiento con las listas resultantes. Los contadores de la lista izquierda son:



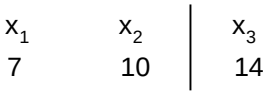
Como se ve ahora los contadores comienzan en 1 y 3 respectivamente y el pivote es el número 14. Aplicando el procedimiento una vez resulta:



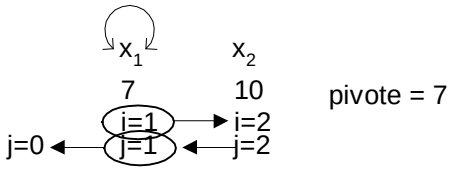
Y repitiendo el procedimiento una vez más se tiene:



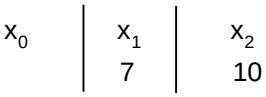
Ahora que los contadores han quedado fijos se debería intercambiar los valores de x_i y x_j, pero como el contador *i* es mayor al contador *j*, no se realiza el intercambio y la lista queda dividida en dos: la izquierda (hasta j=2) con los elementos menores al pivote y la derecha (desde i=3) con los elementos mayores o iguales al pivote:



Cuando una de las listas queda con un solo elemento, como en este caso ocurre con la lista derecha, dicho elemento ya está ordenado, es decir tiene el valor correcto. El vector izquierdo sin embargo tiene dos elementos y aunque vemos que ya están ordenados debemos aplicar el procedimiento al mismo:

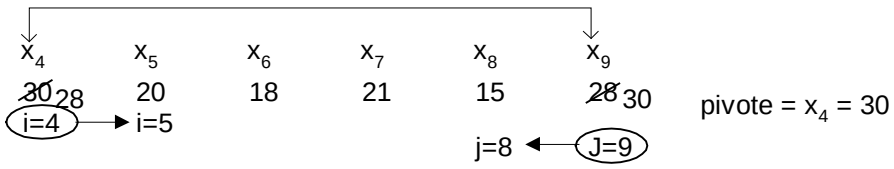


Como se ve en este caso los contadores *i* y *j* llegan al mismo valor y se intercambia el elemento consigo mismo (algo que no se hace en la práctica). Posteriormente (como siempre sucede después de un intercambio) el valor de i incrementa en uno (a 2) y el de j disminuye en uno (a 0). Entonces como *i* tiene un valor mayor a *j* el proceso concluye quedando la lista dividida en dos: el vector izquierdo (hasta j=0) que como vemos en realidad no tiene elementos y el derecho (desde i=2) que queda con un elemento. Al medio queda una lista con un solo elemento y como es único, ese elemento está también ordenado.

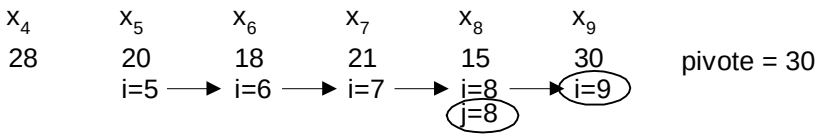


Como todos los vectores quedan con un solo elemento (o ninguno), entonces estos elementos ya están ordenados. De esa manera quedan ordenados todos los elementos que existían en el vector izquierdo resultante de la primera división.

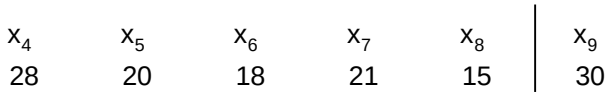
Ahora se aplica el procedimiento al vector derecho de la primera división:



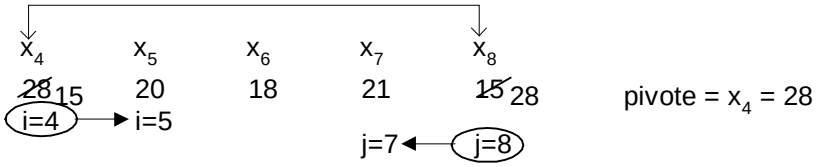
Puesto que *i* sigue siendo menor a *j* se vuelve a aplicar el procedimiento:



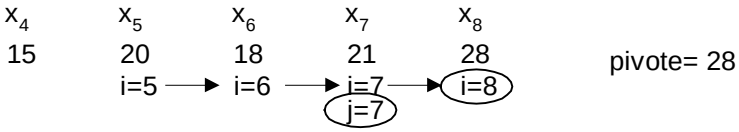
Donde no se realiza ningún intercambio pues *i* es ya mayor a *j*, por lo tanto el proceso concluye y la lista queda dividida en 2:



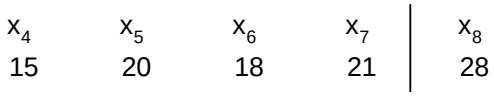
Como la lista derecha tiene un solo elemento, está ya con el valor correcto. Pero como la lista izquierda tiene más de un elemento se aplica el procedimiento a la misma:



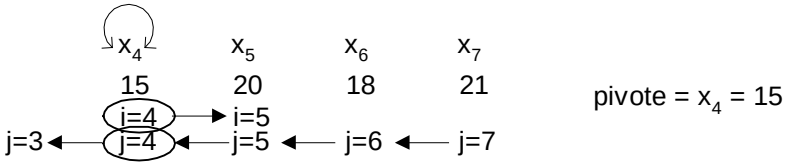
Puesto que *i* sigue siendo menor a *j* se vuelve a aplicar el procedimiento:



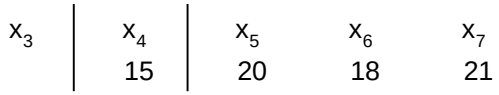
Una vez más no se realiza ningún intercambio pues el contador i es mayor al contador j. Por lo tanto el proceso concluye y la lista queda dividida en dos:



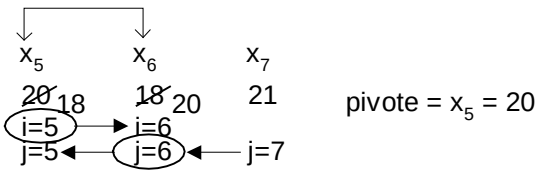
Como la lista derecha tiene un solo elemento (x_8), está ya ordenada. La lista izquierda, sin embargo, tiene más de un elemento, por lo que se vuelve a aplicar el procedimiento:



Ahora *i* es mayor a *j*, por lo que la lista queda dividida en 2: la lista izquierda que no tiene elementos y la lista derecha que tiene tres elementos. Al medio queda un elemento y como es único está ya con el valor correcto:



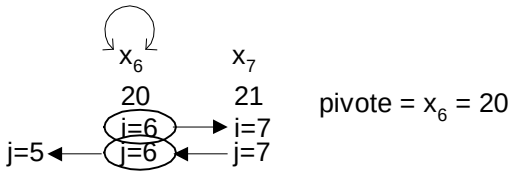
Como la lista derecha tiene más de un elemento, se vuelve a aplicar el procedimiento a la misma:



Entonces la lista queda dividida en 2:

x_5	x_6	x_7
18	20	21

Como la lista izquierda tiene un solo elemento ya está ordenada. Pero como la lista derecha tiene aún 2 elementos se debe aplicar el procedimiento a la misma:



Una vez más la lista queda dividida en 3:

x_5	x_6	x_7
	20	21

Puesto que la lista izquierda no tiene elementos, la derecha sólo uno y al medio queda un elemento, todos las listas tienen un solo elemento y en consecuencia están ya ordenados y como no quedan listas con más de un elemento, todo la lista está ordenada:

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
7	10	14	15	18	20	21	28	30

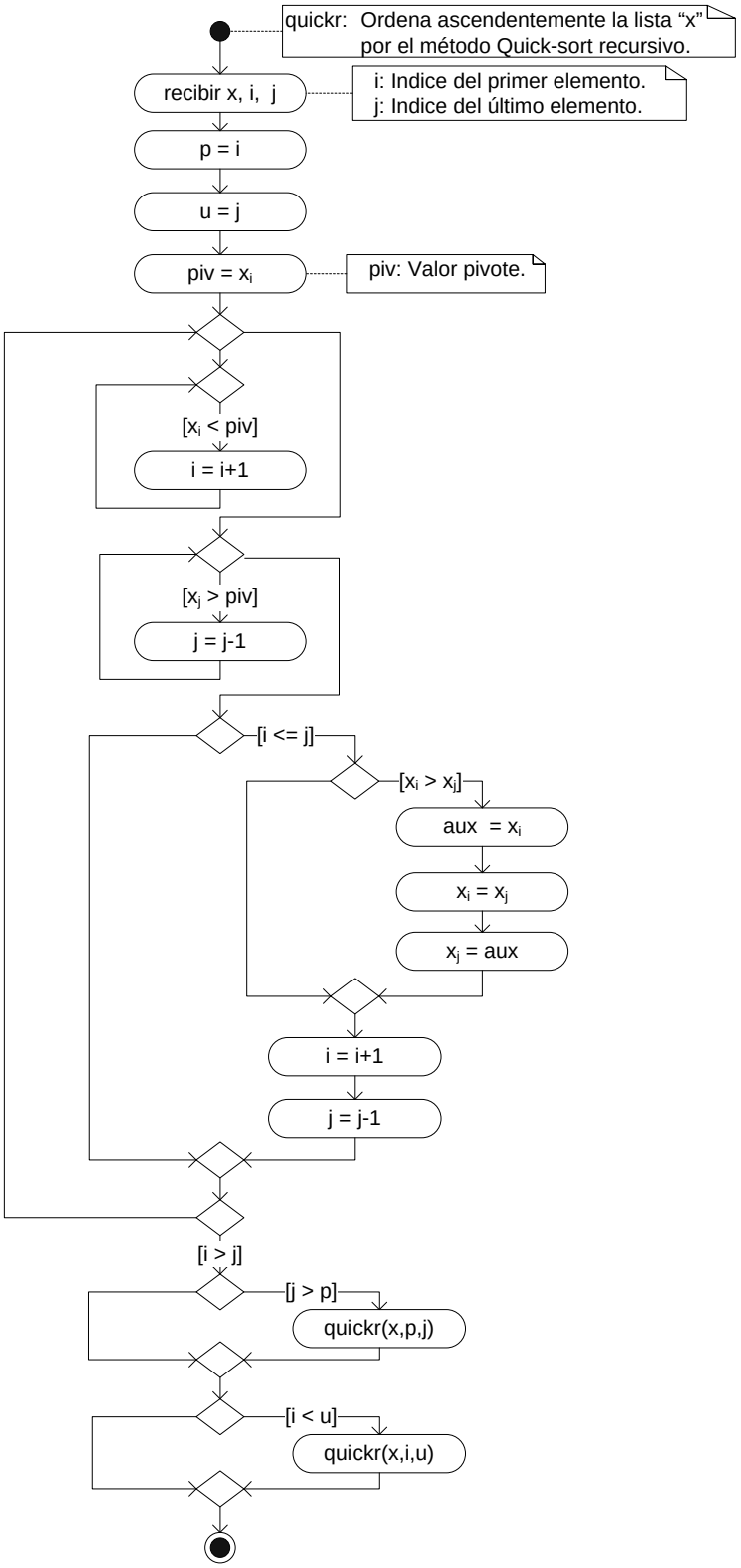
Seguramente al estudiar este ejemplo se preguntarán si no ha existido un error al presentar este método como el más eficiente de todos, porque desde el punto de vista del esfuerzo humano parecería ser todo lo contrario, sin embargo, en todos los casos se repite siempre el mismo procedimiento, algo en lo que los humanos no somos buenos, pero que las computadora realizan con bastante eficiencia. Por el contrario, algo que si consume tiempo de computación, y que en consecuencia es conveniente reducir, son los intercambios de variables, y en ese sentido el método Quick - Sort reduce considerablemente el número de intercambios. Aún en una lista tan pequeña como el del ejemplo, con Quick Sort se requieren 7 intercambios, comparado con los 19 que se requieren con el método de *Burbuja*. A medida que incrementa el número de elementos la diferencia se incrementa y cuando la lista es muy grande (con millones o cientos de millones de elementos) lo que a Burbuja le toma días a Quick Sort le toma minutos.

10.2.1. Algoritmo y código

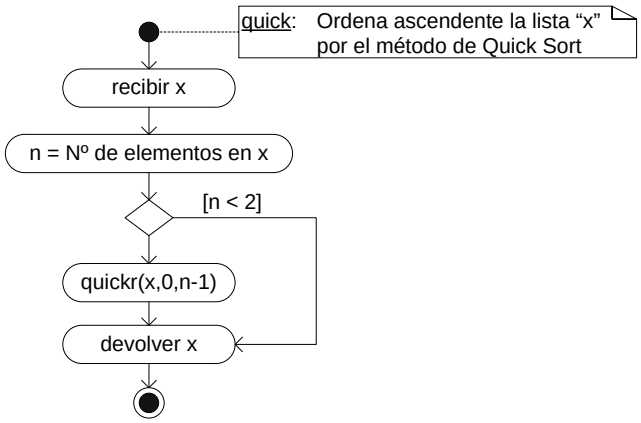
El método Quick-Sort es por naturaleza recursivo, por lo tanto la forma más sencilla de programar el mismo es justamente empleando esta propiedad. Básicamente lo que se hace es programar el procedimiento repetitivo (donde se recorre la lista empleando dos índices hasta que el primer índice es mayor que el segundo) y luego el procedimiento se llama a sí mismo con las

listas resultantes, primero con la lista izquierda y luego con la derecha. El proceso concluye cuando las listas quedan con uno o cero elementos.

El algoritmo del módulo recursivo (que es donde realmente se resuelve el problema) es el siguiente:



Como de costumbre, cuando se resuelve un problema de forma recursiva, se requieren dos módulos: el módulo recursivo que es el que realmente resuelve el problema (y cuyo diagrama es el de la anterior página) y el módulo no recursivo que controla los tipos de datos, las condiciones límites y desde el cual se llama al módulo recursivo para obtener la solución. El diagrama de actividades del módulo no recursivo (donde no se toman en cuenta las condiciones límite ni tipos de datos) es el siguiente:



El código correspondiente a estos diagramas es el siguiente:

```
proc quickr {x i j} {
  set p [copy $i]
  set u [copy $j]
  set piv [lindex $x $i]
  while true {
    while {< [lindex $x $i] $piv} {incr $i}
    while {> [lindex $x $j] $piv} {incr $j -1}
    if {<= $i $j} {
      if {> [lindex $x $i][lindex $x $j]} {
        set aux [lindex $x $i]
        lset $x $i [lindex $x $j]
        lset $x $j $aux
      }
      incr $i
      incr $j -1
    }
    if {> $i $j} break
  }
  if {> $j $p} {quickr $x $p $j}
  if {< $i $u} {quickr $x $i $u}
}
```

```
proc quick x {
  set n [llen $x]
  if {< $n 2} return
  quickr $x 0 [- $n 1]
}
```

Haciendo correr el programa con 10 elementos se obtiene:

```
hecl> set l [generarenteros 10 1 10]
9 3 7 3 7 9 6 4 5 8
hecl> time {quick $l}
0
hecl> puts $l
3 3 4 5 6 7 7 8 9 9
```

Haciendo correr el programa con 2000 elementos se obtiene:

```
hecl> set l [generarenteros 2000 1 2000]
850 1885 1522 319 285 1495 966 98 1495 247 1152 426 894 1801
 1114 70 1645 220 279 480 585 243 640 80 315 1353 1154 1919
1258 1502 911 1477 1456 709 852 66 1798 377 779 370 980 1068
```

```
hecl> time {quick $l}
360
hecl> puts $l
2 4 5 5 7 10 11 12 13 15 15 18 21 21 22 22 22 23 23 24 24 26
 26 27 29 29 30 32 32 33 34 34 34 35 35 36 37 40 41 43 45 46
 47 47 48 49 51 52 52 53 53 54 56 56 57 57 58 58 59 61 61 61
```

Es decir que Quick – Sort sólo requiere 360 milisegundos (ni medio segundo) para ordenar una lista con 2000 elementos: ¡casi siete veces menos que Shell! y ¡casi 94 veces menos que Burbuja!

Con 10000 elementos se obtiene:

```
hecl> set l [generarenteros 10000 1 10000]
176 5406 717 5780 7225 1709 2251 728 9947 4823 712 725 8569
1826 1087 7845 6699 479 6304 8961 6330 9387 6909 3320 7120 1
963 5997 1022 3178 2158 2251 341 7058 2775 1138 6621 1187 64
39 659 1894 1497 5282 3360 2271 6721 7824 780 9696 9476 594
9966 5667 2059 7599 4259 7209 680 2950 2136 9544 6772 623 86
02 5645 2973 671 3339 207 4358 2996 8615 4425 2640 8249 7845
 8405 2137 346 2577 45 1565 6659 1707 2787 4923 41 1747 9768
 8142 1437 1088 8584 5477 3165 4375 9120 1153 619 5351 6760
```

```
hecl> time {quick $l}
2140
hecl> puts $l
1 1 3 4 4 5 6 6 8 9 10 10 11 11 12 13 13 13 14 17 19 21 22 2
2 22 23 23 24 25 27 27 27 28 34 36 36 38 38 39 41 41 42 43 4
5 45 45 46 48 49 52 53 54 55 55 56 56 56 57 57 57 59 60 60 6
2 62 64 64 65 65 66 68 73 74 74 76 78 79 79 80 82 82 83 84 8
4 84 84 87 89 91 92 95 97 98 100 100 100 101 101 101 102 103
 104 104 105 105 106 109 110 110 111 114 114 114 116 117 117
 118 119 121 121 121 122 122 124 124 125 125 126 126 126 126
 129 131 131 131 132 133 134 134 135 136 136 138 141 143 144
 145 147 148 148 150 151 151 152 152 152 152 153 154 155 155
 156 156 157 158 158 159 159 161 164 164 165 166 166 166 167
```

Entonces Quick – Sort requiere un poco más de 2 segundos para ordenar 10000 elementos, mientras que Shell requiere algo más de 19 segundos (casi 9 veces más).

10.3. EJERCICIOS

1. Elabore un módulo que ordene descendentemente una lista empleando el método Shell.

2. Elabore un módulo recursivo, que ordene ascendentemente una lista empleando el método Shell.

3. Elabore un módulo que ordene descendentemente una lista empleando el método Quick-Sort.

4. Elabore un módulo no recursivo que ordene ascendentemente una lista empleando el método Quick-Sort.

11. BÚSQUEDA E INTERCALACIÓN

En este tema se estudian dos métodos de búsqueda y un método de intercalación. La búsqueda se emplea para ubicar un determinado valor dentro de una lista, la intercalación para unir dos o más vectores ordenados sin perder el orden ya existente.

Cuando se trabaja con listas, ubicar un valor implica determinar la posición (índice o dirección de memoria) del elemento que contiene ese valor. De los diferentes métodos de búsqueda que existen estudiaremos los métodos de *búsqueda secuencial* y *búsqueda binaria.*

Hecl cuenta con la instrucción “search”, que busca un elemento en una lista, sin embargo, esta instrucción devuelve el valor buscado, no la posición de dicho valor, lo que es de utilidad muy limitada pues sólo nos permite saber si el valor buscado se encuentra o no en la lista, pero no donde se encuentra. Su sintaxis es la siguiente:

search *lista variable {condición}*

Por ejemplo, para buscar el número 723 en la lista “l”, guardando el resultado en la variable “x” escribimos:

```
search $l x {= $x 723}
```

Si el valor existe nos devuelve “723” y dicho valor se guarda en la variable “x”, si no, no devuelve nada y a la variable “x” se le asigna el último valor de la lista.

Hecl cuenta también con otro comando que permite ubicar a la vez 2 o más valores en una lista: “filter”, sin embargo, y al igual que el caso anterior, dicho comando devuelve los valores de la lista, no sus posiciones, por lo que una vez más es de utilidad limitada: sólo nos permite averiguar cuántos elementos existen en una lista. Su sintaxis es la siguiente:

filter *lista variable {condición}*

Por ejemplo, para buscar el número 11 en la lista “l”, guardando los resultados en la variable “r” escribimos:

```
filter $l r {= $r 11}
```

Si en la lista existen 3 valores iguales a 11, “filter” devuelve una lista con esos tres valores ({11 11 11}). Si no existe ningún valor igual al buscado no devuelve nada. A diferencia de “search”, “filter” siempre asigna a la variable de búsqueda (“r” en el ejemplo) el último elemento de la lista.

A propósito “Hecl” cuenta también con una instrucción para ordenar listas: “sort”, la cual además es muy rápida, sin embargo, dicha instrucción trata todos los valores a ordenar como texto, devolviendo una lista ordenada alfabéticamente, por lo que no devuelve el orden correcto en caso de valores numéricos. Su sintaxis es la siguiente:

sort lista

11.1. MÉTODO DE BÚSQUEDA SECUENCIAL

Este método, conocido también como búsqueda lineal, se emplea cuando los elementos no están ordenados. Al estar los datos desordenados, no se puede aplicar ningún algoritmo para acelerar la búsqueda, pues el elemento a buscar puede estar en cualquier posición, por lo que la búsqueda secuencial lo

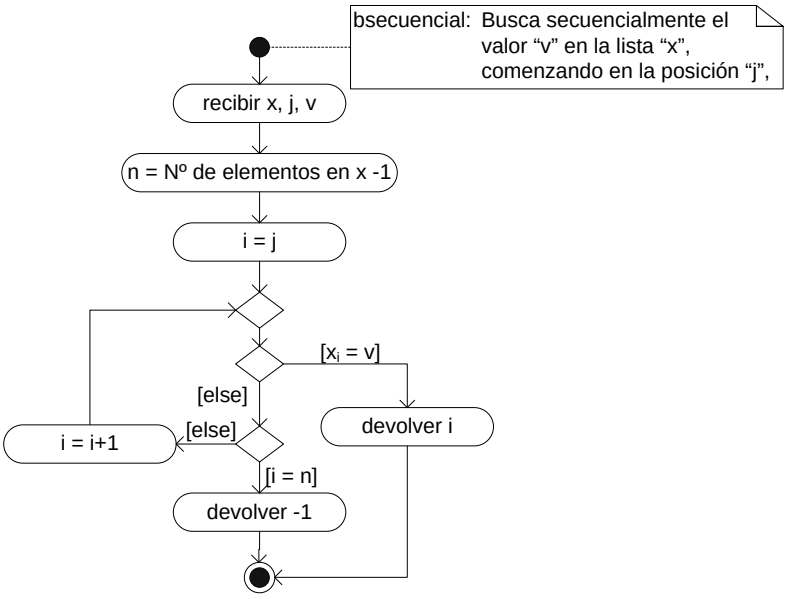
único que se hace es recorrer la lista, elemento por elemento, hasta ubicar el valor buscado o hasta que ya no existen más elementos en la lista.

Como en una lista pueden existir valores duplicados, una vez ubicado un elemento, se puede continuar la búsqueda una posición después, para ubicar la posición de otro de los valores en caso de que existieran.

A pesar de su simplicidad, el método de búsqueda secuencial es un método de utilidad práctica, porque mucha de la información existente, como el texto de los documentos, se encuentra desordenada, no existiendo otra alternativa para la búsqueda que la secuencial.

11.1.1. Algoritmo y código

El algoritmo del método se presenta en la siguiente figura, en el mismo si el valor buscado no se encuentra en la lista se devuelve -1, es decir una posición imposible (porque como se sabe el menor índice en una lista es cero).



El código respectivo es:

```
proc bsecuencial {x j v} {
   set n [- [llen $x] 1]
   for {set i $j} {<= $i $n} {incr $i} {
     if {= [lindex $x $i] $v} {return $i}
   }
   return -1
}
```

Para probar el módulo se genera a continuación una lista con 100 números enteros comprendidos entre 1 y 100:

```
hecl> set x [generarenteros 100 1 100]
8 7 60 97 57 61 33 85 3 93 76 27 76 57 72 45 76 24 18 97 75
64 8 60 75 1 62 19 97 94 29 42 62 12 42 54 91 97 52 23 50 26
 97 48 34 14 36 76 87 19 84 46 42 86 21 95 44 7 97 52 55 51
35 97 86 13 90 5 66 18 76 42 14 9 93 53 14 3 76 61 18 84 70
79 78 93 7 44 81 53 55 92 38 48 80 37 20 5 72 75
```

Entonces se busca el número 33, que como vemos está en la séptima posición, es decir en la posición número 6:

```
hecl> bsecuencial $x 0 33
6
```

Y como vemos se obtiene el resultado correcto. Ahora se puede volver a buscar en la lista, pero comenzando desde la posición 7 (6+1) para verificar si existe o no otro número 33:

```
hecl> bsecuencial $x 7 33
-1
```

Como el resultado es -1, entonces no existe otro valor igual a 33 en la lista (lo que se puede corroborar fácilmente viendo la lista).

Para contar con un mayor número de elementos repetidos se genera a continuación una lista con 100 elementos comprendidos entre 1 y 50:

```
hecl> set x [generarenteros 100 1 50]
5 9 8 38 5 37 31 2 12 34 42 8 28 26 34 36 14 17 26 35 41 18
39 15 7 37 43 49 6 10 24 2 20 34 33 26 5 15 44 6 11 42 8 3 1
5 36 37 43 47 1 22 18 43 29 37 13 18 22 43 44 17 38 44 45 27
 33 42 23 4 30 48 12 30 37 24 41 2 31 10 17 42 41 11 32 18 1
5 21 19 1 20 3 17 36 25 42 23 39 40 17 49
```

En esta lista se busca el número 5 (que aparece 3 veces) hasta que “bsecuencial” devuelve -1:

```
hecl> bsecuencial $x 0 5
0
hecl> bsecuencial $x 1 5
4
hecl> bsecuencial $x 5 5
36
hecl> bsecuencial $x 37 5
-1
```

Con lo que se comprueba que el método encuentra todos los valores de la lista. Por supuesto en lugar de ejecutar las instrucciones una por una, es posible escribirlas dentro de un ciclo iterativo:

```
hecl> set i 0; set p {};
while true {
  set i [bsecuencial $x $i 5]
  if {= $i -1} break
  lappend $p [ copy $i]
  incr $i
}
hecl> puts $p
0 4 36
```

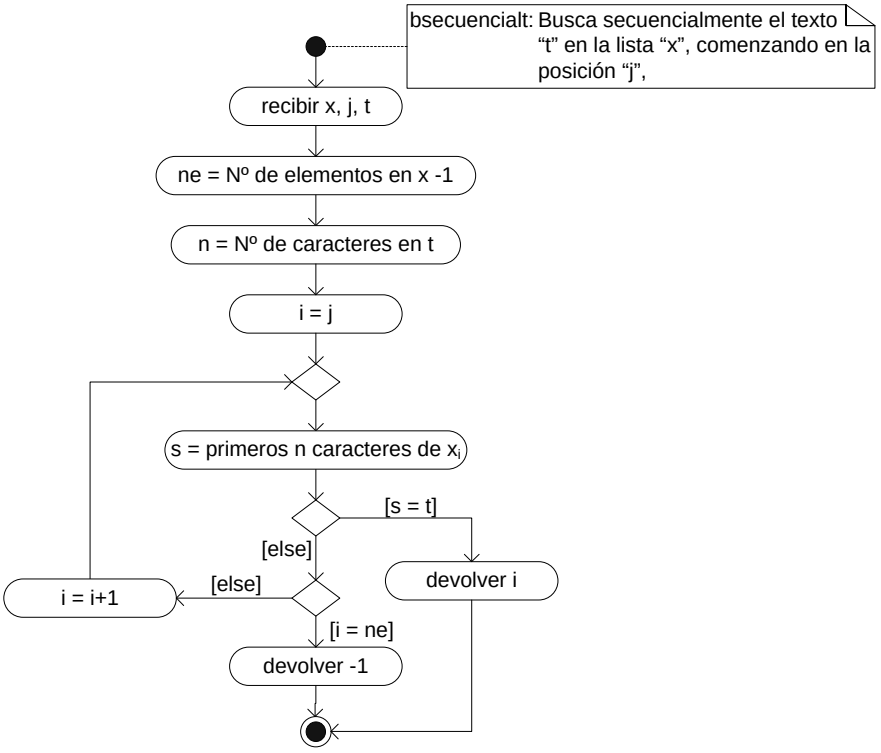
Que por supuesto devuelve los mismas resultados, sólo que ahora los mismos se encuentran en una lista.

11.1.2. Algoritmo y código con cadenas

Aunque el método de búsqueda secuencial (al igual que los otros métodos) no cambia con el tipo de dato que se busque, cuando se trabaja con cadenas, frecuentemente sólo se quiere buscar una parte de la misma. Por ejemplo cuando se busca un nombre con frecuencia sólo se escriben las primeras letras del mismo (y no el nombre completo). En esos casos el método debe ser capaz de encontrar el primer nombre cuyas primeras letras coincida con las letras escritas.

Para ello el algoritmo debe ser ligeramente modificado, de manera que el valor buscado se compare sólo con las primeras letras de los elementos y no con el valor completo (como sucede con el anterior algoritmo).

El algoritmo que toma en cuenta dicha modificación es el siguiente:



Siendo el código respectivo:

```
proc bsecuencialt {x j t} {
   set ne [- [llen $x] 1]
   set n [- [strlen $t] 1]
   for {set i $j} {<= $i $ne} {incr $i} {
     set s [strrange [lindex $x $i] 0 $n]
     if {eq $s $t} {return $i}
   }
   return -1
}
```

Observe que para compara cadenas no se emplea el símbolo "=", sino el operador "eq", igualmente para verificar si dos cadenas son diferentes se emplea el operador "ne" en lugar de "!=". Si se quiere averiguar si una cadena es mayor, menor o igual a otra se emplea la función "strcmp" que devuelve 1 si la primera cadena es mayor a la segunda, -1 si la primera es menor que la segunda y 0 cuando son iguales.

Para probar el módulo creamos una lista con algunos nombres:

```
hecl> set l {Carlos Juan Carmen Jose Nancy Ramiro Maria Mar
lene Alicia Gabriela}
Carlos Juan Carmen Jose Nancy Ramiro Maria Marlene Alicia Ga
briela
```

Para buscar los nombres que comienzan con "Rami" escribimos:

```
hecl> bsecuencialt $l 0 Rami
5
hecl> bsecuencialt $l 6 Rami
-1
```

Es decir que “Rami” aparece una sola vez en la séptima posición. Con “Car” obtenemos lo siguiente:

```
hecl> bsecuencialt $l 0 Car
0
hecl> bsecuencialt $l 1 Car
2
hecl> bsecuencialt $l 3 Car
-1
```

Es decir que “Car” aparece dos veces, una en la posición 1 y otra en la posición 3.

Sin embargo, para probar el método de búsqueda con listas más grandes es conveniente contar con algún módulo que permita generar valores aleatorios, pues introducirlos manualmente, como en el ejemplo, no resulta práctico sobre todo cuando se trabaja con decenas, centenas o miles de datos.

Uno de los datos que aparece con mayor frecuencia en la práctica son los nombres de las personas, por lo que es conveniente contar con un módulo para este fin. Para ello debemos escribir primero algunos módulos que devuelvan nombres y apellidos al azar, es decir, debemos contar con algún “dominio” de nombres y apellidos.

Un módulo que permite generar un nombre femenino es el siguiente:

```
proc GenerarNombreF {} {
  global NombreF
  if {catch {llen $NombreF}} {
    set NombreF {ANA AMALIA ANDREA ADELA
      ARIANA ANABEL ADRIANA BETTY BELINDA CARMEN CARLA CECILIA
      CINTIA DANITZA DIANA DORA DANIELA ELENA ELMI ESTER ELY
      EDMI FABIOLA FERNANDA CABRIELA GLADYS GEORGINA GLORIA
      HILDA ILSEN ISABEL JANETH JANINA JULIA JUDITH KARINA
      LIDIA LILIANA LIZETH LUPE LUZ LOURDES MARIA MARIANA
      MARITZA MARCELA MIRIAN MAGDALENA MARTA NOEMI NANET NOELIA
      OLIMPIA OFELIA PATRICIA PAULINA PAMELA ROMINA ROSSEMARY
      RUTH ROSA SANDRA SUSANA SELENA SILVINA SCARLET TERESA
      TALIA VANIA VERONICA VIVIANA VIRGINIA XIMENA YARA YANESI
      ZULEMA
    }
  }
  return [lindex $NombreF [round [* [random] 75]]]
}
```

En este módulo se declara como global la variable “NombreF” y si la misma no ha sido aún inicializada, lo que se comprueba capturando con “catch” el error que se produce al tratar de leer el número de elementos de una variable no declarada, se le asignan los 76 nombres de la lista. Se emplea una variable global para no tener que asignar a la variable los mismos valores cada vez que se llama al módulo.

Luego para devolver al azar uno de esos nombres, se genera un número aleatorio entero comprendido entre 0 y 75 (con round [* [random] 75]), constituyéndose dicho número en el índice de la lista, así si el número generado es 0, el módulo devuelve el primer elemento de la lista: “ANA”, si es 5, el sexto elemento “ANABEL” y así sucesivamente. Por supuesto, mientras mayor sea el número de nombres en la lista más aleatorio es el nombre devuelto.

Llamando al módulo unas cuantas veces se obtiene:

```
hecl> GenerarNombreF
JULIA
hecl> GenerarNombreF
GLORIA
hecl> GenerarNombreF
PATRICIA
hecl> GenerarNombreF
BETTY
hecl> GenerarNombreF
MARCELA
hecl> GenerarNombreF
MIRIAN
hecl> GenerarNombreF
MIRIAN
hecl> GenerarNombreF
PAULINA
hecl> GenerarNombreF
NOEMI
hecl> GenerarNombreF
ELY
hecl> GenerarNombreF
LIDIA
hecl> GenerarNombreF
GLADYS
hecl> GenerarNombreF
AMALIA
hecl> GenerarNombreF
GLADYS
```

Que como se puede ver son nombres obtenidos al azar.

De manera similar se crean otros dos módulos: uno para los nombres masculinos y otro para los apellidos:

```
proc GenerarNombreM {} {
  global NombreM
  if {catch {llen $NombreM}} {
    set NombreM {ALEJANDRO ALFREDO ALVARO
      ANTONIO ANGEL ALCIDES ALBERTO BENIGNO BETO BORIS CARLOS
      CESAR CECILIO DAVID DANIEL ENRIQUE ERNESTO EDWIN EDSON
      ELIAS FERNANDO FABRICIO FREDDY FELIX FELICIANO GASTON
      GILMAR GONZALO GERSON GUIDO HENRY HERNAN HORACIO IGNACIO
      ISAAC IVES JAVIER JHON JOSE JUAN JORGE JAIME LUIS
      LIMBERTH MARCELO MICHEL MAURICIO MAX NELSON NESTOR
      ORLANDO PEDRO PABLO PAUL RICHARD ROBERTO RICARDO RAFAEL
      RENE SANDRO SANTOS TITO ULISES VICENTE VICTOR VLADIMIR
      WILSON WILLY WILFREDO YVANOX YAMIL ZENON
    }
  }
  return [lindex $NombreM [round [* [random] 71]]]
}
```

```
proc GenerarApellido {} {
  global Apellido
  if {catch {llen $Apellido}} {
    set Apellido {AGUILAR ALBINO ALVAREZ
      ARANCIBIA AVILA ANDRADE ARANDIA ARAUZ ARDUZ ARIAS BARRON
      BALLIVIAN BARRIENTOS BEJARANO BEDOYA BELLIDO BELTRAN
      BENAVIDEZ BORDA BOBARIN BORJA CABA CAMPOS CARDOZO CARMONA
      CARREON CARRASCO CARRILLO CARVALLO CASTAÑOS CHOQUERIVE
      CEPEDA CONDORI COCA CORTEZ CRESPO CRUZ DALENCE
```

```
        DAVALOS DAVEZIES DAVILA DAZA DELGADILLO DULON DURAN
        ECHALAR ECHEVERRIA EID ESCALANTE ENRIQUEZ ESPADA FERNANDEZ
        FLORES FONSECA FRIAS FUERTES GALLARDO GALVAN GARCIA GOMEZ
        GORENA GRIMALDOS HERRERA HINOJOSA HURTADO IBAÑEZ IBARRA
        ILLANES INCHAUSTI JALDIN JARA JIMENEZ KAWANO KOMAREK
        LAGRAVA LAGUNA LAIME LARA LAZCANO LEAÑO LEYTON LLANOS
        LLAVE LLOBET MAMANI MARTINEZ MEDINA MENDIENTA MEZA MOLINA
        MUÑOZ NAVA NOGALES NOYA NAVARRO ORGAZ ORIAS OROZCO
        ORTIZ ORTUSTE OTONDO OVANDO PACHECO PALACIOS PADILLA
        PEMINTEL PERALTA PEREIRA PEREZ PLANTARROSA POPPE PRADEL
        QUEVEDO QUINTEROS QUIÑONES QUIROGA QUIROZ QUENTA
        RADA RAMIREZ RENDON REYNAGA REYNOLDS RENTERIA RIOS RIVERA
        ROCAVADO ROCHA ROJAS ROMERO RUIZ SAAVEDRA SALAMANCA
        SALAS SALAZAR SANCHEZ SANDI SANTOS SERRANO STUMVOLL
        TARDIO TABOADA TERAN TIRADO TOLEDO TORRES TORRICO
        TORREJON UGRINOVIC URIONA URQUIDI URQUIZU URRIOLAGOITIA
        URIETA VACA VACAGUZMAN VALDEZ VALENCIA VARGAS VALVERDE
        VENTURA VELASQUEZ VILLAVICENCIO VILLEGAS WAYAR WILLIAMS
        YAÑEZ YUGAR ZAMORA ZARATE ZELAYA ZEGADA ZEBALLOS ZULETA
        ZORRILLA ZURITA
    }
  }
  return [lindex $Apellido [round [* [random] 175]]]
}
```

Con estos módulos podemos crear ahora un módulo que genere un nombre completo con dos apellidos y uno o dos nombres:

```
proc GenerarNombre {} {
  if {< [random] 0.5} {set n [GenerarNombreM]
    if {> [random] 0.5} {set n [append $n " "[GenerarNombreM]]}
  } else {set n [GenerarNombreF]
    if {> [random] 0.5} {set n [append $n " "[GenerarNombreF]]}}
  return [append [GenerarApellido]" "[GenerarApellido]" "$n]
}
```

En este módulo se genera primero un número aleatorio (comprendido entre 0 y 1) y si el mismo es menor a 0.5 se genera un nombre masculino, caso contrario se genera uno femenino. En cada uno de los casos anteriores se vuelve a generar un número aleatorio y si el mismo es mayor a 0.5 se genera otro nombre, caso contrario se queda con un solo nombre.

Haciendo correr el módulo unas cuantas veces se obtiene:

```
hecl> GenerarNombre
RIOS BOBARIN ELMI DORA
hecl> GenerarNombre
BEDOYA CARDOZO NANET
hecl> GenerarNombre
ORGAZ ENRIQUEZ FERNANDO TITO
hecl> GenerarNombre
LAGUNA OROZCO BORIS ROBERTO
hecl> GenerarNombre
FERNANDEZ QUIÐONES LIZETH
hecl> GenerarNombre
BARRON BEDOYA SCARLET
```

Ahora, podemos crear un módulo que empleando el anterior genere “n” nombres completos aleatorios:

```
proc GenerarNombres { n } {
  set x {}
  for {set i 1} {<= $i $n} {incr $i} {
    lappend $x [GenerarNombre]
  }
  return $x
}
```

Haciendo correr este módulo con 50 se obtiene:

```
hecl> GenerarNombres 50
{ARANDIA PALACIOS ELMI} {ROCHA ORTIZ MAX} {OTONDO PERALTA YU
ANOX} {ZULETA CAMPOS FABRICIO WILFREDO} {CEPEDA GOMEZ JAVIER
 SANTOS} {WILLIAMS BORDA ISAAC} {CHOQUERIVE MAMANI RENE DAVI
D} {CARDOZO ROCAVADO VERONICA} {URRIOLAGOITIA MEZA PABLO REN
E} {ROCAVADO ALVAREZ XIMENA} {PACHECO VACAGUZMAN GERSON VLAD
IMIR} {WAYAR REYNAGA SELENA} {URIETA QUINTEROS ALFREDO} {RAM
IREZ TORREJON CARMEN} {BORJA SANCHEZ MARITZA} {ECHEVERRIA BO
RJA PEDRO} {FLORES ZEBALLOS MARIANA} {ZORRILLA TORRICO NOELI
A ILSEN} {OROZCO SAAVEDRA ANGEL FERNANDO} {URIETA OROZCO GAS
TON} {REYNAGA TIRADO GERSON} {URIETA RENTERIA ERNESTO} {RIOS
 CAMPOS ERNESTO MAX} {HURTADO TOLEDO LILIANA} {CARRILLO SALA
S PATRICIA LUPE} {KOMAREK HINOJOSA ANDREA DIANA} {SANTOS BEN
AVIDEZ WILLY ALFREDO} {GOMEZ DALENCE RICHARD ALBERTO} {RADA
TORRICO DIANA MARIA} {ESCALANTE COCA CESAR EDSON} {QUENTA TO
RRES RUTH JUDITH} {TIRADO QUIROZ GILMAR MICHEL} {ROCHA ZARAT
E ISABEL ARIANA} {ESPADA ROCHA ROSA} {NAVA URQUIDI SELENA RU
TH} {ROCAVADO MEZA MARIA JANETH} {ESPADA ZAMORA ALCIDES FERN
ANDO} {POPPE CABA HERNAN DAVID} {RENDON STUMVOLL MARIA} {AND
RADE FUERTES TALIA} {DAVALOS MEDINA LOURDES} {ARANCIBIA ALVA
REZ ALBERTO BETO} {GALVAN DAVILA NANET FERNANDA} {BOBARIN PO
PPE ELENA DORA} {CARMONA MAMANI NELSON} {DAVEZIES VALDEZ BET
O ROBERTO} {NOGALES BORJA ALBERTO} {ZEGADA MAMANI ELMI PAULI
NA} {RUIZ ARANCIBIA GERSON} {ESCALANTE JIMENEZ PEDRO}
```

Que como se puede ver son nombres completos aleatorios.

Ahora se puede probar el método de búsqueda secuencial con 1000, 2000 o un mayor número de elementos. Por ejemplo se pueden generar 1000 elementos y buscar todos los nombres que comiencen con “ORT” (por supuesto, al ser valores aleatorios, los resultados no serán los mismos al hacer correr el módulo otra vez o en otra computadora):

```
hecl> set x [GenerarNombres 1000]
{CARRASCO ROCAVADO JANINA} {CONDORI MOLINA MAGDALENA SUSANA}
 {VALVERDE VACA LIMBERTH} {TERAN INCHAUSTI MARIA DORA} {UGRI
NOVIC JIMENEZ GERSON FABRICIO} {PADILLA WAYAR ARIANA LOURDES
} {POPPE ROCHA NESTOR ANTONIO} {ZEBALLOS ZORRILLA YARA} {TIR
ADO DAVILA LOURDES} {ECHALAR DAZA ALEJANDRO HORACIO} {ZELAYA
 IBARRA GLADYS VANIA} {LARA RIOS FABRICIO} {LAGUNA NAVARRO A
```

```
hecl> bsecuencialt $x 0 ORT
36
hecl> bsecuencialt $x 37 ORT
71
hecl> bsecuencialt $x 72 ORT
75
hecl> bsecuencialt $x 76 ORT
297
hecl> bsecuencialt $x 298 ORT
456
hecl> bsecuencialt $x 457 ORT
485
hecl> bsecuencialt $x 486 ORT
705
hecl> bsecuencialt $x 705 ORT
```

```
734
hecl> bsecuencialt $x 735 ORT
-1
```

En este caso entonces existe en la lista 8 nombres que comienzan con "ORT" y los mismo se encuentran en las posiciones 37, 72, 76, 298, 457, 486, 706 y 735.

11.2. MÉTODO DE BÚSQUEDA BINARIA

El método de búsqueda binaria se emplea cuando los elementos del vector están ordenados. Por lo tanto, si la lista no está ordenada debe ser ordenada previamente por uno de los métodos estudiados en el anterior tema (o alternativamente se puede emplear "sort" si es una lista de cadenas).

Los módulos que se elaborarán en este tema están pensados en buscar elementos en listas ordenadas ascendentemente, por supuesto, sólo se requiere una pequeña modificación para buscar en listas ordenadas descendentemente.

En este método se compara el valor buscado con el elemento central del vector. Si el valor buscado es igual al elemento central el proceso concluye, caso contrario se divide el vector en dos y se repite el procedimiento en la mitad donde es probable que se encuentre el valor buscado. Este procedimiento se repite (dividiendo el vector en dos en cada iteración) hasta que el valor es encontrado o hasta que la mitad donde se realiza la búsqueda queda sin elementos.

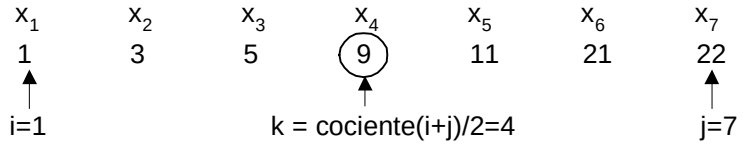
Se sabe en cuál de las mitades se encuentre el valor buscado comparándolo con el elemento central: Si es mayor al elemento central entonces se encuentra en la mitad derecha (o superior) y si es menor en la mitad izquierda (o inferior).

Cuando en el vector existen dos o más elementos con el valor buscado, el método de búsqueda binaria ubica uno de esos elementos, pero no garantiza que dicho elemento sea el primero. Por consiguiente, una vez ubicado el elemento, es necesario recorrer el vector hacia atrás hasta que el primer elemento diferente al valor buscado.

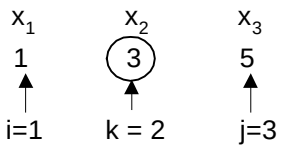
Para comprender mejor el procedimiento que se sigue en el método se ubicará el número 5 en el siguiente vector:

x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	3	5	9	11	21	22

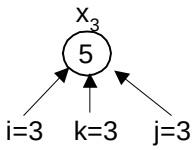
En el proceso se emplean 3 variables: *"i"* para el índice del primer elemento; *"j"* para el índice del último elemento y *"k"* para el índice del elemento central. El valor de *"k"* se calcula como el cociente de la división de *i+j* entre 2:



Como el elemento central es mayor al valor buscado (5) se repite el procedimiento en la mitad izquierda. El último índice a ser tomado en cuenta *(j)* es 3 (k-1):

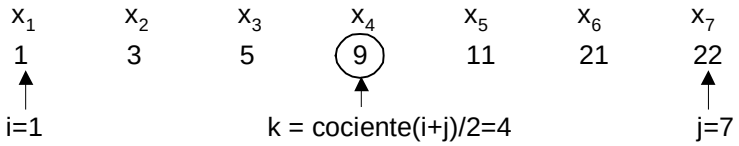


Ahora el elemento central es menor al valor buscado (5), entonces se repite el procedimiento en la mitad derecha de este vector, en esta mitad sólo queda un elemento:

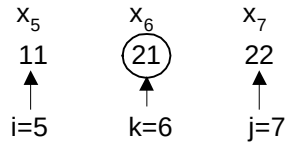


Y como ese elemento es igual al valor buscado el valor buscado ha sido encontrado en la posición 3 (el valor de *k*).

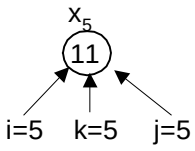
Ahora empleemos el método para ubicar el número 11. Como siempre el proceso comienza tomando en cuenta todo el vector:



Puesto que el elemento central es menor al valor buscado (11) se repite el procedimiento en el vector derecho:



Dado que el elemento central es mayor al valor buscado (11) se repite el procedimiento con el vector izquierdo, que tiene un solo elemento:



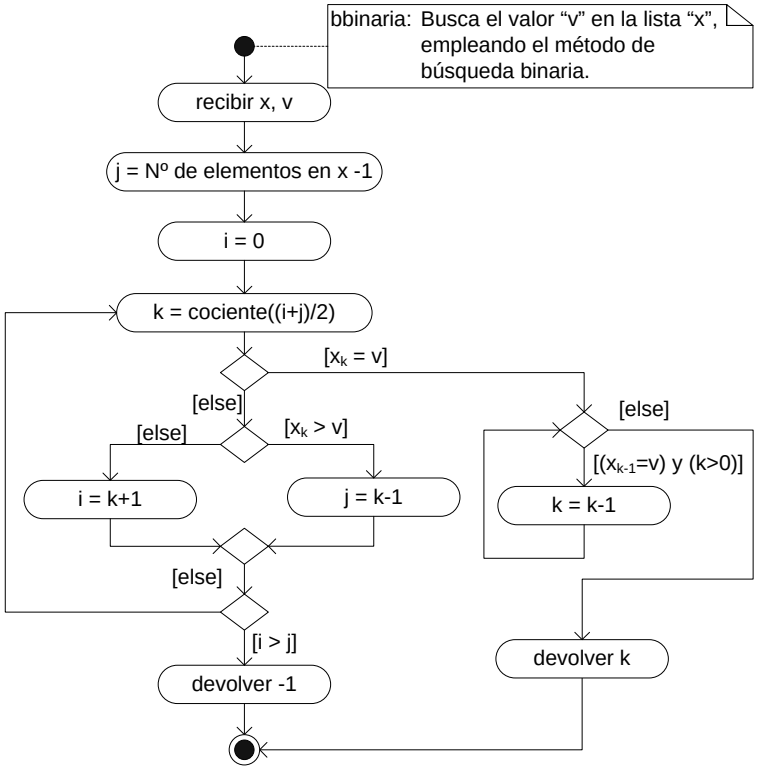
Y como el mismo es igual al valor buscado el proceso concluye habiéndose ubicado el valor en la posición 5 (el valor de *k*).

Veamos que sucede cuando en el vector no existe el valor buscado. Por ejemplo si en lugar del número 11 se busca el número 12. Aplicando el procedimiento se llega al mismo vector con un solo elemento (x_5) y como el elemento central es menor al valor buscado (12) se debería proseguir la búsqueda en el vector derecho, es decir "i" debería tomar el valor de "k" más 1 (5+1=6), pero entonces ocurre que "i" es mayor que "j" (6>5), lo que nos indica que ya no existen más elementos en la sub-lista derecha.

Por lo tanto se sabe que el valor buscado no se encuentra en la lista cuando al realizar la búsqueda no quedan más elementos en las sub-listas, es decir cuando el valor de la variable *"i"* es mayor que el de la variable *"j"*.

11.2.1. Algoritmo y código

Con el ejemplo queda claro que el método de búsqueda binaria puede ser programado fácilmente de manera recursiva, pero también puede ser programado de manera iterativa, tal como se muestra en el siguiente algoritmo:



Cuyo código es el siguiente:

```
proc bbinaria {x v} {
  set j [- [llen $x] 1]
  for {set i 0} {<= $i $j} {} {
    set k [/ [+ $i $j] 2]
    if {= [lindex $x $k] $v} {
      while {and [= [lindex $x [- $k 1]] $v] [> $k 0]} {incr $k -1}
      return $k
    } else {
      if {> [lindex $x $k] $v} {
        set j [- $k 1]} else {set i [+ $k 1]}
    }
  }
  return -1
}
```

Para probar este módulo, podemos generar una lista con 1000 números comprendidos entre 1 y 1000, ordenarla con Quick-Sort y luego buscar en dicha lista algunos números:

```
hecl> set x [generarenteros 1000 1 1000]
10 171 473 694 696 39 363 815 904 494 99 122 951 166 377 943
 484 92 799 104 26 792 3 536 58 238 311 776 575 550 169 998
15 150 396 671 425 196 713 16 772 586 872 443 276 500 146 82
```

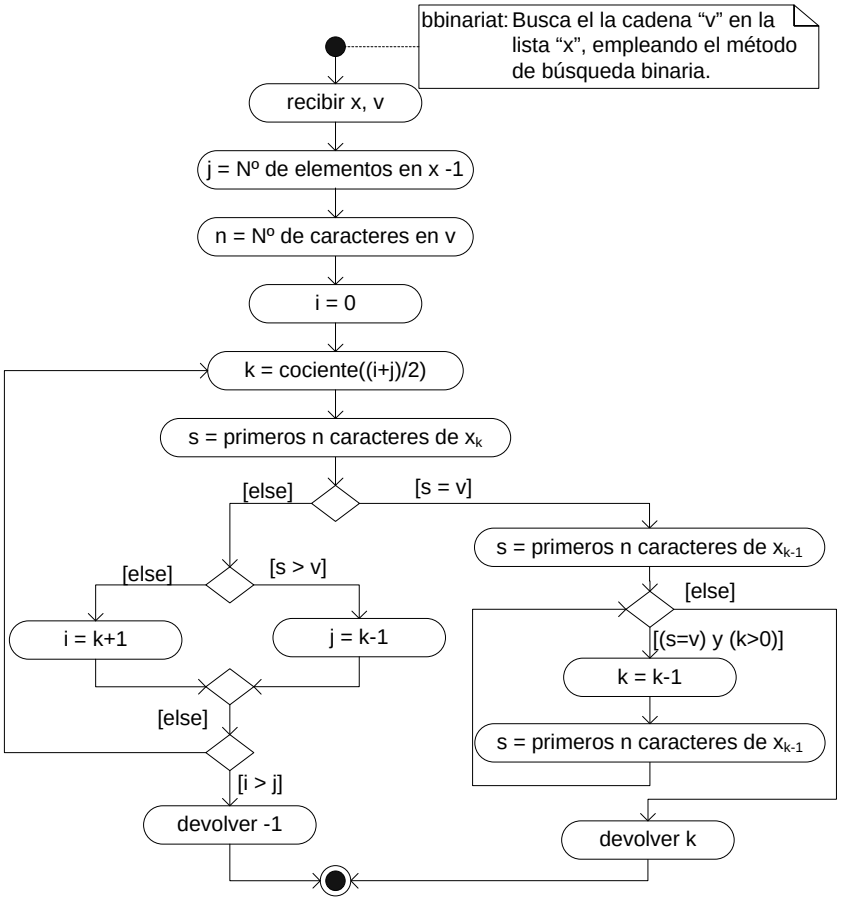
```
hecl> quick $x
hecl> puts $x
3 5 5 6 9 10 11 13 15 15 15 16 16 17 19 19 19 19 20 21 22 22
 23 23 24 26 26 28 29 31 32 32 33 34 35 35 35 36 39 39 39 40
 41 41 42 43 43 43 44 44 45 46 47 48 49 51 55 56 56 57 58 58
```

```
hecl> bbinaria $x 600
588
hecl> bbinaria $x 51
55
hecl> bbinaria $x 848
841
hecl> bbinaria $x 982
982
hecl> bbinaria $x 997
-1
hecl> bbinaria $x 1
-1
```

Vemos entonces que existen en la lista los número 600, 51, 848 y 982 (en las posiciones 588, 55, 841 y 982), pero que no existen los números 997 y 1 (por supuesto y como ya se dijo, al ser números aleatorios, estos resultados difieren de corrida en corrida y de máquina a máquina).

11.2.2. Algoritmo y código con cadenas

Al igual que sucede con el método de búsqueda secuencial, si bien el método no cambia, sin importar qué es lo que se esté buscando, en el caso de las cadenas (texto) se debe hacer una ligera modificación, principalmente porque es probable que lo que se busque sólo sea parte de la cadena y no la cadena completa. El algoritmo para este fin es el siguiente:



Siendo el código respectivo el siguiente:

```
proc bbinariat {x v} {
  set j [- [llen $x] 1]
  set n [- [strlen $v] 1]
  for {set i 0} {<= $i $j} {} {
    set k [/ [+ $i $j] 2]
    set s [strrange [lindex $x $k] 0 $n]
    if {eq $s $v} {
      set s [strrange [lindex $x [- $k 1]] 0 $n]
      while {and [eq $s $v][> $k 0]} {
        incr $k -1
        set s [strrange [lindex $x [- $k 1]] 0 $n]
      }
      return $k
    } else {
      if {> [strcmp $s $v] 0} {
        set j [- $k 1] } else {set i [+ $k 1]}
    }
  }
  return -1
}
```

Para probar este código, se pueden generar listas de nombres con el módulo desarrollado en el anterior acápite, pero como la lista tiene que estar ordenada, es necesario emplear luego algún método de ordenación.

Por supuesto, como en este caso se trata de cadenas, es posible emplear la instrucción “sort” de Hecl, no obstante, para recordar lo aprendido en el anterior tema y practicar con cadenas, es más productivo modificar los métodos estudiados, de manera que ordenen cadenas (texto) en lugar de números.

Como ejemplo, se ha modificado el método Quick-Sort, de la siguiente manera:

```
proc quickrs {x i j} {
  set p [copy $i]
  set u [copy $j]
  set piv [lindex $x $i]
  while true {
    while {< [strcmp [lindex $x $i] $piv] 0} {incr $i}
    while {> [strcmp [lindex $x $j] $piv] 0} {incr $j -1}
    if {<= $i $j} {
      if {> [strcmp [lindex $x $i][lindex $x $j]] 0} {
        set aux [lindex $x $i]
        lset $x $i [lindex $x $j]
        lset $x $j $aux
      }
      incr $i
      incr $j -1
    }
    if {> $i $j} break
  }
```

```
    if {> $j $p} {quickrs $x $p $j}
    if {< $i $u} {quickrs $x $i $u}
  }
proc quicks x {
    set n [llen $x]
    if {< $n 2} return
    quickrs $x 0 [- $n 1]
}
```

Ahora generamos una lista con 500 nombres:

```
hecl> set x [GenerarNombres 500]
{TORRES GALVAN DANIEL PEDRO} {YAÐEZ GORENA JORGE} {SANCHEZ T
ABOADA LILIANA} {URQUIZU BALLIVIAN ALCIDES PABLO} {REYNOLDS
CARREON JULIA} {NAVARRO CABA EDSON} {QUEVEDO ZEBALLOS FELIX}
 {ENRIQUEZ VELASQUEZ NELSON} {VACAGUZMAN ESPADA ELY} {SAAVED
```

Ordenamos la lista con “quicks”:

```
hecl> quicks $x
```

Y para verificar luego los valores encontrados la mostramos con sus índices respectivos a la izquierda:

```
hecl> set i 0
0
hecl> foreach n $x {puts "$i : $n"; incr $i}
0 : AGUILAR WILLIAMS YAMIL MAURICIO
1 : ALBINO LAIME ROSA
2 : ALBINO QUINTEROS GUIDO JUAN
3 : ALBINO SERRANO ANGEL
4 : ALVAREZ FRIAS MARCELA ELY
5 : ALVAREZ OTONDO ALCIDES
6 : ALVAREZ RENTERIA ANTONIO GONZALO
7 : ALVAREZ REYNOLDS WILLY
8 : ALVAREZ ZELAYA SANDRA SUSANA
9 : ANDRADE CHOQUERIVE ARIANA OFELIA
10 : ANDRADE ILLANES ROSSEMARY DORA
```

```
489 : ZORRILLA CRESPO LUZ PATRICIA
490 : ZORRILLA DAVEZIES YANESI LUPE
491 : ZORRILLA INCHAUSTI CECILIO PAUL
492 : ZORRILLA PEREZ SCARLET SCARLET
493 : ZORRILLA STUMVOLL ZULEMA XIMENA
494 : ZORRILLA TABOADA MAGDALENA
495 : ZULETA RUIZ ALEJANDRO
496 : ZULETA TOLEDO ROSSEMARY VERONICA
497 : ZULETA URQUIZU ENRIQUE YVANOX
498 : ZURITA RUIZ GABRIELA NANET
499 : ZURITA URQUIZU ZENON
```

Ahora ubicamos algunos nombres y verificamos si las posiciones son correctas:

```
hecl> bbinariat $x "ALVAREZ Z"
8
hecl> bbinariat $x "ANDRADE ILL"
10
hecl> bbinariat $x "ZORRILLA IN"
491
hecl> bbinariat $x "ZORRILLA TA"
494
hecl> bbinariat $x "ZULETA UR"
497
hecl> bbinariat $x "ZURITA R"
498
hecl> bbinariat $x "MESA"
-1
```

```
hecl> bbinariat $x "NINA"
-1
hecl> bbinariat $x "ZUBIETA"
-1
```

Que como vemos devuelve las posiciones correcta y -1 en caso de no encontrarse el nombre en la lista.

11.3. INTERCALACIÓN

La intercalación es la operación por la cual se unen dos listas ordenadas para formar una tercera, igualmente ordenada, con los elementos de ambas. Por supuesto las listas podrían ser simplemente añadidas y posteriormente ordenadas con uno de los métodos de ordenación estudiados, sin embargo, el proceso de ordenación consume más tiempo que el de intercalación, además dicha operación implica la repetición innecesaria de operaciones (se vuelven a ordenar listas que ya estaban ordenadas).

La intercalación puede ser empleada también para ordenar listas muy grandes. En ese caso se ordenan segmentos de la lista empleando uno de los métodos de ordenación y luego se intercalan los segmentos ordenados, consiguiendo así la lista ordenada.

Este mismo procedimiento puede ser empleado también para mejorar la eficiencia de los métodos directos (principalmente burbuja y selección): se divide el proceso de ordenación en listas pequeñas (de unos 50 elementos), y a medida que las mismas son ordenadas, con el método directo, se intercalan en un vector resultante, que finalmente contendrá toda la lista ordenada.

En la intercalación los elementos de las listas se van añadiendo a la lista resultante de forma tal que la misma siempre queda ordenada. Para ello simplemente se compara un elemento de una de las listas con otro elemento de la otra (comenzando los primeros elementos de las dos listas) y se añade a la lista resultante el menor de ellos. Esta operación se repite (sin tomar en cuenta los elementos añadidos) hasta que no quedan más elementos en una de las listas. Cuando esto sucede, los elementos de la otra lista (la que quedó con elementos) se añaden al final de la lista resultante.

Por ejemplo si queremos intercalar las siguientes listas:

i	1	2	3	4	5	6
x	6	15	23	30	34	40
y	5	10	25	26	30	

Comenzamos comparando los primeros elementos de ambas listas (6 con 5) y puesto que 5 es menor que 6, añadimos este valor a la lista resultante (z):

i	1	2	3	4	5	6
x	6	15	23	30	34	40
y	~~5~~	10	25	26	30	
z	5					

Entonces comparamos 6 con 10 y puesto que 6 es menor que 10, dicho valor es añadido a la lista:

i	1	2	3	4	5	6
x	~~6~~	15	23	30	34	40
y	~~5~~	10	25	26	30	
z	5	6				

Ahora comparamos 15 con 10 y dado que 10 es menor a 15 añadimos 10 a la lista resultante:

i	1	2	3	4	5	6
x	~~6~~	15	23	30	34	40

y	~~5~~	~~10~~	25	26	30	
z	5	6	10			

Proseguimos de esa manera: comparamos 15 con 25 (añadimos el número 15), luego 23 con 25 (añadimos el número 23), 30 con 25 (añadimos el número 25), 30 con 26 (añadimos el número 26), 30 con 30 (como 30 no es menor a 30, añadimos el número 30 de la segunda lista). En este punto se acaban los elementos de la lista *"y"*:

i	1	2	3	4	5	6	7	8	9	10	11
x	~~6~~	~~15~~	~~23~~	30	34	40					
Y	~~5~~	~~10~~	~~25~~	~~26~~	~~30~~						
Z	5	6	10	15	23	25	26	30			

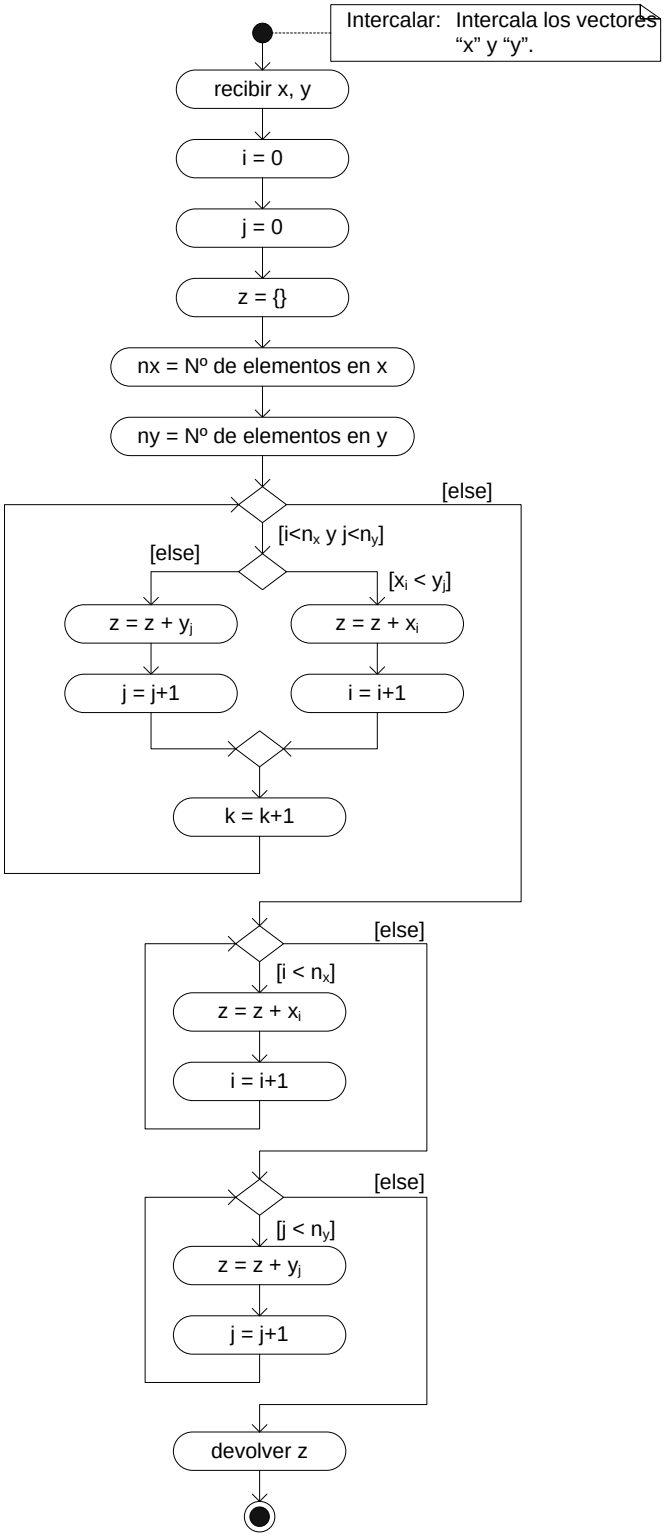
Entonces los elementos restantes de la lista *"x"* (desde x_4 hasta x_6) son añadidos al final de la lista resultante, con lo que obtenemos la lista intercalada *("z")*:

I	1	2	3	4	5	6	7	8	9	10	11
X	~~6~~	~~15~~	~~23~~	~~30~~	~~34~~	~~40~~					
Y	~~5~~	~~10~~	~~25~~	~~26~~	~~30~~						
Z	5	6	10	15	23	25	26	30	30	34	40

11.3.1. Algoritmo y código

El algoritmo para intercalar dos listas (ordenadas ascendentemente), se muestra en el diagrama de actividades de la siguiente página y el código elaborado en base al mismo es el siguiente:

```
proc intercalar {x y} {
  set i 0
  set j 0
  set z {}
  set nx [llen $x]
  set ny [llen $y]
  while {and [< $i $nx][< $j $ny]} {
    if {< [lindex $x $i][lindex $y $j]} {
      lappend $z [lindex $x $i]
      incr $i
    } else {
      lappend $z [lindex $y $j]
      incr $j
    }
  }
  while {< $i $nx} {
    lappend $z [lindex $x $i]
    incr $i
  }
  while {< $j $ny} {
    lappend $z [lindex $y $j]
    incr $j
  }
  return $z
}
```



Primero se probará el programa con dos listas de 10 elementos:

```
hecl> set x [generarenteros 10 1 10]
8 4 9 4 8 9 6 7 4 1
hecl> set y [generarenteros 10 1 10]
5 7 1 9 4 3 3 1 4 2
hecl> quick $x
hecl> quick $y
```

```
hecl> puts $x
1 4 4 4 6 7 8 8 9 9
hecl> puts $y
1 1 2 3 3 4 4 5 7 9
hecl> intercalar $x $y
1 1 1 2 3 3 4 4 4 4 4 5 6 7 7 8 8 9 9 9
```

Y dado que el programa funciona correctamente, podemos probar con listas más grandes, por ejemplo de 2000 elementos:

```
hecl> set x [generarenteros 2000 1 3000]
1714 779 858 497 426 931 917 1988 2486 2847 1353 827 2299 96
 1798 2716 2582 1404 1644 1594 2833 2837 670 2109 971 2260 7
22 1107 810 2567 1895 2943 1806 1034 1241 2011 2634 1045 244
8 1910 2653 1517 1212 2444 99 411 130 880 26 977 1894 2805 1
```

```
hecl> set y [generarenteros 2000 1 3000]
1468 1112 1315 1201 2977 2772 1190 2722 1366 1579 1062 2332
2556 2824 1770 1105 2246 1077 1320 371 1911 87 1624 977 1322
 1331 107 2904 136 2436 1952 1798 2772 440 245 1294 2814 265
9 2529 62 467 1838 2480 1016 365 231 347 335 1005 1373 674 2
```

```
hecl> quick $x
hecl> quick $y
hecl> time {intercalar $x $y}
47
hecl> intercalar $x $y
1 1 2 3 3 3 3 3 3 4 4 6 6 8 8 9 10 12 12 13 13 14 14 14 18 1
9 19 19 20 22 22 22 22 23 23 25 26 26 27 28 28 30 30 31 31 3
2 33 33 34 37 38 38 41 42 44 45 45 45 46 48 48 49 50 51 53 5
3 53 54 54 56 56 57 57 58 60 61 61 62 63 64 64 64 64 65 68 6
```

Como se puede observar, el proceso de intercalación de las dos listas de 2000 elementos sólo requiere 47 milisegundos, tiempo considerablemente inferior al de ordenación.

11.4. EJERCICIOS

1. Elabore un módulo que empleando el método de selección, devuelva en una lista todas las posiciones en las que se encuentra un determinado valor en la misma.

2. Elabore los módulos que sean necesarios para generar aleatoriamiente lista con direcciones de calles.

3. Elabore un módulo recursivo para buscar un valor numérico en una lista ordenada ascendentemente, empleando el método de búsqueda binaria.

4. Elabore un módulo recursivo para buscar una cadena en una lista ordenada ascendentemente, empleando el método de búsqueda binaria.

5. Elabore un módulo para ordenar listas de cadenas por el método de burbuja.

6. Elabore un módulo para ordenar listas de cadenas por el método de selección.

7. Elabore un módulo para ordenar listas de cadenas por el método de inserción.

8. Elabore un módulo para ordenar listas de cadenas por el método Shell.

9. Elabore un módulo que intercale dos listas de cadenas (ordenadas ascendentemente).