FUNDAMENTOS DE PROGRAMACIÓN CON RUBY

Luis Eduardo Muñoz G

Universidad Tecnológica de Pereira
Facultad de Ingenierías
Ingeniería de Sistemas y Computación

Luis Eduardo Muñoz Guerrero
Autor
Profesor Titular
Universidad Tecnológica de Pereira

Miguel Ángel López Fernández
Editor literario, compilador y corrector
Estudiante de Ingeniería de Sistemas y Computación
Universidad Tecnológica de Pereira

1ª Edición, Pereira-Risaralda. Julio de 2022

ISBN

Créditos de edición e impresión

FUNDAMENTOS DE PROGRAMACIÓN CON RUBY

AGRADECIMIENTOS

A mi hijo Camilo, por siempre recordarme el significado de la palabra tranquilidad.

A mi padre Luis Muñoz, quien ha sido un ejemplo sencillez a lo largo de mi vida.

Agradezco de manera especial al estudiante Miguel Ángel López Fernández, por sus aportes y trabajos en el desarrollo del libro.

INTRODUCCIÓN

Este documento presenta los conceptos fundamentales de la programación imperativa desarrollada con el lenguaje de Ruby. Se hablará sobre temas relacionados con los tipos de datos, las funciones y estructuras de control, hasta el manejo de estructuras de datos como *arreglos, listas*, *pilas*, *colas y deques*, así como introducir el paradigma de la programación orientada a objetos (conocida también como POO)

Se propone como lectura para personas que quieran iniciarse en la programación con la ayuda de un lenguaje de alto nivel, es decir, mucho más fácil de comprender. El documento sirve como base conceptual para experimentar a futuro con temas más avanzados. Por otra parte, cada sección está dividida en un componente de teoría y conceptos, definición de sintaxis y ejemplos de prueba, y otra que usa ejercicios que recopilen la información analizada mediante su desarrollo y análisis. Ciertos apartados disponen de ejercicios prácticos para ayudar al lector a retroalimentarse.

Al final se menciona la bibliografía usada como fundamento teórico para desarrollar toda la información aquí presente, es de recordar que todo es con base a la documentación oficial que soporta el lenguaje Ruby, además de otras referencias como videos y textos poco académicos que colaboran al desarrollo.

TABLA DE CONTENIDO

LISTA DE TABLAS

LISTA DE ILUSTRACIONES

INTRODUCCIÓN AL LENGUAJE DE RUBY

Se dará una breve contextualización y explicación sobre el lenguaje de programación Ruby, el campo de acción al cual pertenece, su historia y particularidades. Se indicarán también las instrucciones para poder instalar a manejar la sintaxis Ruby

1 El lenguaje de programación Ruby

1.1 ¿Qué es?

Ruby es un lenguaje de programación que se destaca por su facilidad en el aprendizaje, se puede implementar en varios campos tecnológicos siendo este gran beneficio para abordar problemas desde un modelo funcional, imperativo, abstracción a objetos, entre muchos otros.

Fue diseñado principalmente para mejorar en los programadores facultades como la *productividad*, *mantenimiento* y *comprensión* de sus códigos. Esta última se enfatizó hasta el punto de hacer de Ruby un lenguaje divertido de aprender, incluso, su creador piensa que en el mundo de la tecnología *los humanos somos los masters, ellos (las computadoras) los esclavos*, por lo que todo debería ser en pro del entendimiento humano para controlar una máquina, no al contrario.

La filosofía de Ruby se generaliza en tres grandes características: la primera es el principio de *bajo asombro*, la segunda es la fácil disposición de recursos para un enfoque más pragmático en la escritura de código, y la tercera es la transparencia del lenguaje. El principio de *bajo asombro* (POLA, por sus siglas en inglés *principle of low atonishment)* indica que el lenguaje de programación arbitrario debe de comportarse según lo esperado por quien lo usa para así evitar tener momentos de confusión y asombro-impacto. La segunda indica la capacidad de Ruby para proveer de servicios que permitan al programador tener a su disposición múltiples vías de solución a un problema en específico, ya sea por medio de algún paradigma de programación[1], funciones que reduzcan el tiempo invertido, sintaxis comprensible, adaptación de funcionalidades de otros lenguajes, simpleza, entre muchas otras. Como tercera, se entiende como transparencia del lenguaje la capacidad de explicarse por sí mismo, esto es debido gracias a la particularidad que tiene Ruby de ser de *alto nivel*[2]

1.2 Particularidades de Ruby

Ruby tiene un gran listado de características que lo vuelven en una gran herramienta para la solución de los problemas tecnológicos del día a día.

Algunas de las características más sobresaliente son (1):

---

[1] Los paradigmas de programación son, de manera general, modelos conceptuales que indican la forma en cómo se dan las instrucciones a la maquina (computador). Algunos de los paradigmas más conocidos y utilizados son el funcional (encargado de indicar que instrucciones hacer, esto generalmente por medio de funciones ya diseñadas), el imperativo (encargado de indicar como hacer las instrucciones, es decir, explicar a detalle la orden a ejecutar), el proposicional (encargado de ejecutar instrucciones partiendo de estructuras lógicas formales, es decir, basándose en la lógica de proposiciones).

[2] Se conoce como lenguaje de alto nivel aquel que se maneja y expresa de manera similar al lenguaje humano. Algunos ejemplos además de Ruby son Python, ALGOL, Basic, Erlang, etc.

- Lenguaje expresado y trabajado completamente con la programación orientada a objetos, soportando herencia y simulación de herencia múltiple con *mixins* (estructuras propias del lenguaje), así como el uso de la *meta programación*.

- Es de tipado dinámico, es decir, que no es obligatorio expresar el tipo de dato a utilizar. También incorpora mucho *azúcar* sintáctico en muchas de funciones (sin contar la sintaxis propia del lenguaje que también se le atribuye el término mencionado).
- Sintaxis que permite acercarse al dominio de un problema, así como permitir la flexibilidad y conveniencia observada en otros programas como Python.
- Estructuras propias basadas en lenguajes mucho más antiguos como Smalltalk (relación de semántica), Perl (relación de sintaxis).
- Dinamicidad en el manejo de variables (tanto para clases como para métodos)
- Recolector de basura, lo que indica que el uso de la memoria en tiempo de ejecución, conocida como memoria de HEAP es gestionada (su uso y liberación) de manera automática.
- Manejo de excepciones, facultad necesaria para realizar pruebas o tests en un código determinado.
- Manejo de Hilos
- Actualmente existen muchas implementaciones del lenguaje Ruby con características propias como relación de cambio a otros lenguajes como C o Java.
- Tiene su propio sistema de gestión de paquetes conocido como RubyGems.
- Es un lenguaje compilado justo en el tiempo de ejecución (*Just-in-time compilation* por sus siglas en inglés). Esta es una técnica que utiliza las dos formas de traducción de código conocidas, la compilación y la interpretación, donde primero se compila el código volviéndolo a código de bits (bytecode en inglés) para luego justo al momento de iniciar la ejecución del programa, se inicie al mismo tiempo la interpretación (traducción) de del código de bits anterior al código de máquina.

1.3 Ruby en la actualidad

1.3.1 ¿Se usa Ruby hoy en día?

Como ya se ha mencionado, Ruby es un lenguaje de propósito general trayendo consigo múltiples modelos de programación, imperativa, funcional, lógica, POO, entre otras. Es por ello que la capacidad que provee para solucionar y desarrollar múltiples ideas es bastante grande.

A continuación, se comparten algunos de los proyectos documentados en la web los cuales están hechos principalmente en Ruby (2):

- *Hackety Hack*, un programa que enseña a programar usando Ruby.

- *Sass* (Syntactically Awesome Style Sheets), una extensión en el trabajo con *CSS* (Cascading Style Sheet) usando variables en la expresión de reglas de estilo. Fue originalmente escrito en Ruby, y actualmente se encuentra bajo C/C++.
- *Homebrew*, el instalador de paquetes utilizado en el documento para descargar Ruby.
- *Discourse* es un aplicativo web que funciona como listero de mensajes, foros de discusión y salas de chat. Está escrito en Ruby con ayuda de otras herramientas como bases de datos Postgres o el framework Rails (también escrito en Ruby).

- Sinatra es un DSL (Domain Specific Language, o lenguaje de dominio específico) creado netamente en Ruby y dedicado a resolver problemas relacionados con el desarrollo web de una manera más eficaz.

Existen docenas y docenas de proyectos elaborados con Ruby, unos por completo y otros como ayuda en aspectos muy específicos.

Lo anterior son algunos ejemplos de lo que se puede lograr con Ruby, que en su mayoría están relacionados con el desarrollo Web. Por otra parte, no se debe aislar la idea de que este lenguaje no pueda ser usado para crear aplicaciones de escritorio, juegos en 2D, inteligencias artificiales y machine learning, entre otras.

1.4 Descarga, instalación y configuración de interfaz de trabajo

La instalación de Ruby puede hacerse de varias maneras algunas por medio de la descarga del paquete o archivo que contiene al lenguaje de Ruby, como también se pueden usar sistemas de gestión de paquetes desde la terminal de cada equipo. Para este caso se usará la última opción, la cual proporciona ciertos beneficios como resumir la instalación y configuración del lenguaje en el equipo a través de solo un conjunto de instrucciones.

1.4.1 Procesos de descarga e instalación (Mac)

Conociendo el método, los pasos a seguir son los siguientes:

- Descargar un gestionador de paquetes como por ejemplo Homebrew

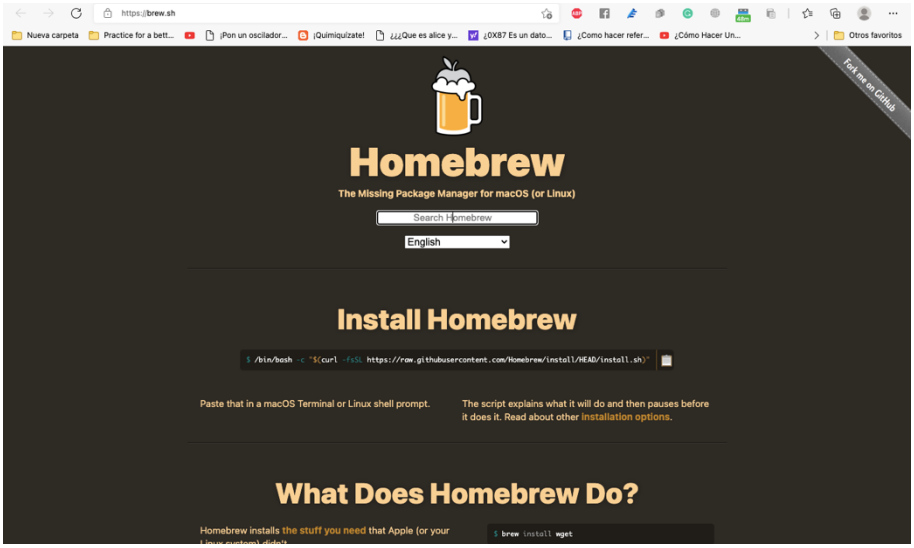


Ilustración 1. Página web de la herramienta Homebrew

Fuente: The Missing Package Manager for macOS (or Linux) — Homebrew

- Se deben de seguir las instrucciones mencionadas allí (ubicadas bajo el comando de terminal debajo del título *Install Homebrew*)
- Otra manera de instalarlo es por medio del terminal del equipo digitando el siguiente comando:

$	brew -v

- Luego de tener la herramienta gestionadora de paquetes, se debe de correr el siguiente comando en terminal

$	brew install ruby

- Posteriormente, se debe de configurar la variable de entorno[3] con el siguiente comando en terminal

$	echo 'export PATH="/usr/local/opt/ruby/bin:$PATH"' >>~/.bash_profile

- Para finalizar se puede verificar si la instalación fue correcta aplicando los siguientes comandos:

$	ruby -v //Para verificar la versión
$	ruby -e "puts 'Hello World'" //Para correr una instrucción con Ruby

[3] Las variables de entorno es un concepto atribuido a ***variables*** pertenecientes al sistema operativo que corre dentro de un equipo de cómputo las cuales permiten almacenar, acceder y obtener cierta información de este, como rutas de archivos, funciones y especificaciones. Por ejemplo, la variable de entorno PATH almacena las rutas de ciertos archivos a ejecutar.

Como resultado se debe de obtener:

```
miguellopez@MacBook-Air-de-Miguel ~ % ruby -v
ruby 2.6.3p62 (2019-04-16 revision 67580) [universal.x86_64-darwin20]
miguellopez@MacBook-Air-de-Miguel ~ % ruby -e "puts 'hello world!'"
hello world!
miguellopez@MacBook-Air-de-Miguel ~ %
```

Ilustración 2. Ejecución de comandos propios de Ruby

La instalación solo se describe para sistemas operativos IOS (como por ejemplo los portátiles Mac), más, sin embargo, al final del documento en la sección de *recursos* se dejarán algunos links que proporcionarán información al respecto.

1.5 Primeras interacciones con el entorno

Nota aclaratoria

En el presente documento se desarrollarán ejercicios que propicien el uso del lenguaje de Ruby a partir del IDE RubyMine por JetBrains. Es decisión propia de cada lector sobre que IDE's utilizar.

1.5.1 Hola mundo desde Ruby

Se comienza con la creación de una de las frases más célebres en el mundo de la programación. "Hello, ¡world!", un pequeño programa nacido en 1978 con el documento "The C programming language".

Ahora bien, dependiendo del IDE's que el lector haya escogido, cree un nuevo archivo en una carpeta cualquiera (puede ser una destinada al desarrollo de este documento) y nómbrelo como *helloWorld.rb* (*rb* es la extensión de Ruby). Luego de abrir el archivo se debe de digitar allí lo siguiente:

```
1 puts "Hello World!! from RubyMine"
```

Se debe de ejecutar en la línea anterior (es decir correr el archivo) y obtener la siguiente salida:

```
$ Hello World!! from RubyMine
```

¡¡¡Con ello se ha creado el primer código en Ruby!!!

COMO ESCRIBIR CÓDIGO EN RUBY

Se hablará sobre la estructura de la sintaxis utilizada en Ruby para la escritura de pequeños programas, así como analizar temas fundamentales para la programación como los tipos de datos y operadores, estructuras de control y flujo, funciones, estructuras de datos. Por último, se explicarán temas más avanzados como la programación orientada a objetos.

2 Estructura de código

2.1 Tipos de operadores

Al igual que las matemáticas en donde se utilizan ciertas operaciones representadas por cierta simbología como lo son la suma (+), resta (-), multiplicación (*), división (/), o incluso operaciones más complejas como derivadas () e integrales (), la programación también maneja su propio conjunto de símbolos comúnmente llamados operadores. El nombre permite conocer cuál será la función de dicha simbología, y en este caso se trata de poder ejecutar operaciones entre uno, dos o más datos entre sí.

A continuación, se presentarán los operadores aritméticos propios de la computación actual, y que con los cuales se nos permite relacionar matemática y lógicamente las instrucciones que son escritas por medio de una computadora.

2.1.1 Operadores aritméticos

Estos operadores (por su nombre, aritmética) permiten representar las operaciones elementales realizables con números. Dichas operaciones son las conocidas sumas, resta, multiplicación y división.

Nombre	Suma	Resta	Multiplicación	División
Símbolo	+	-	*	/

Tabla 1. Operadores aritméticos

Ruby utiliza generalmente la notación infija para poder realizar las operaciones descritas por estos operadores. Por ejemplo:

1 **+** 1
2 **-** 100
1 ***** 56
90.2 **/** 2.5678

Si bien se recuerda, en las matemáticas se cuenta con ciertos de niveles de jerarquía de operadores, incluso de otros símbolos como el paréntesis, la llave y el corchete. De manera similar ocurre en la programación Ruby. Dichos niveles de jerarquía indicaban el orden de importancia (y de ejecución) de las operaciones dadas en una expresión, por ejemplo:

(4 *(3*(3+90) / (78*1034)))

(20/45*(89+17*49))

Los niveles de jerarquía anteriormente dichos son de la siguiente manera:

Orden
Multiplicación (*)
División (/)
Suma (+)
Resta (-)

Tabla 2. Simbología de operado

2.1.2 Operadores lógicos

Al igual que los operadores de la suma o de la resta existen expresiones literarias que determinan las comparaciones usadas en la lógica de proposiciones como el *Y* (AND) o el *O* (OR). Este grupo indica el cumplimiento de una condición u operación lógica, muy comunes en estructuras de control como condicionales e iteradores.

Nombre	Símbolo	Función
NOT	!,~, not	Niega un dato booleano (dato verdadero o falso)
AND	&&, and	Evalúa dos elementos a la vez. Los dos deben de tener el mismo valor (verdadero o falso), caso contrario devuelve falso.
OR	||, or	Escoge el valor entre dos datos, ya sea verdadero o falso.

Tabla 3. Operadores lógicos

Ejemplo

Partiendo de la simbología anterior, se puede presentar el siguiente código:

```
1  print("NOT: < true -> (Not) -> #{!true} > || < false -> (Not) -> #{not false} >\n")
2  print("AND: < true (And) false -> #{true && false} > || < false (And) false -> #{false and false} >\n")
3  print("OR: < true (or) false -> #{true || false} > || < false (And) false -> #{false or false} >\n")
```

La ejecución del código es:

```
$  NOT: < true -> (Not) -> false > || < false -> (Not) -> true >
   AND: < true (And) false -> false > || < false (And) false -> false >
   OR: < true (or) false -> true > || < false (And) false -> false >
```

Se pueden presentar varias posibilidades en cuanto al uso de operadores lógicos con respecto a una salida booleana (verdadero o falso), por ende, se proporcionan las tablas de verdad de cada uno de los operadores según sean los casos posibles:

A	B	A && B
Verdad	Verdad	Verdad
Verdad	Falso	Falso
Falso	Verdad	Falso
Falso	Falso	Falso

Tabla 4. Tabla de verdad para el operador lógico y binario and (&&, &).

A	B	A || B
Verdad	Verdad	Verdad
Verdad	Falso	Verdad
Falso	Verdad	Verdad
Falso	Falso	Falso

Tabla 5. Tabla de verdad para el operador lógico y binario or (||, |).

A	~A
Verdad	0
0	Verdad

Tabla 6. Tabla 5. Tabla de verdad para el operador lógico y binario not (~).

Por otra parte, se tienen iguales representaciones de los operadores lógicos para un pequeño apartado que son las operaciones binarias. La representación es:

Nombre	Símbolo
NOT	!,~
AND	&
OR	|

Tabla 7. Operadores bianarios

Las operaciones dadas a partir de estos operadores son basadas en la lógica de circuitos y sistemas binarios.

Nota: En este documento no se comentará respecto a la teoría detrás de los operadores binarios, más si se recomienda al lector consultar al respecto.

Cabe tener en cuenta que los resultados mostrados para los operadores lógicos, incluyendo la clasificación de las tablas de verdad, son los mismos para los operadores binarios, con la diferencia de que al ser sistemas binarios estos se manejan con 0's y 1's representando *falso* y *verdadero* respectivamente.

2.1.3 Operadores de comparación

Como su nombre lo indica, es la simbología encargada de comparar dos o más datos. Su representación es:

Nombre	Símbolo	Función
Igual que	==	Determina si dos elementos son iguales
No igual o distinto de	=!	Determina si dos elementos son diferentes
Menor que	<	Si un elemento a es menor que un elemento b
Menor o igual que	<=	Determina si es un elemento a es menor o igual que un elemento b
Mayor que	>	Si un elemento a es mayor que un elemento b
Mayor o igual que	>=	Determina si es un elemento a es mayor o igual que un elemento b

Tabla 8. Operadores de comparación

Estos operadores son utilizados principalmente en la evaluación de condiciones, de frases o expresiones en las que se pueda determinar si algo es verdad o no. Por esta razón se pospondrá su ejemplificación hasta llegar al tema de condicionales.

2.1.4 Niveles de procedencia de los operadores

A continuación, se presenta la siguiente tabla que indica el nivel de procedencia de los operadores descritos anteriormente.

Jerarquía de Operadores	
Paréntesis	( )
+/-	signo
^	Potencia
*** y /**	Producto y división
Div	Division entera
Mod	Residuo
+ y -	Suma y resta
+ o &	Concatenación
>,<,>=,<=, <>,=	Relacionales
Not	Negación
Y	And
O	Or

Tabla 9. Niveles de procedencia de operadores

Fuente: jerarquía de operandos en programación - Bing images

2.2 Tipos de datos

Los seres humanos pueden clasificar toda la información que les llega a niveles sumamente singulares, es decir, al punto de dividir y categorizar cada parte en lo más mínimo posible. Por ejemplo, a partir de un texto se puede destacar que este se puede componer de letras, que juntas conforman palabras y consecutivamente forman oraciones, ideas, y párrafos completos; en ellos pueden estar inmersos números; caracteres especiales, inclusive imágenes; en general, un gran conjunto de información representativa de un objeto o idea. Todo ello permite desglosar un cúmulo de información en partes trabajables y analizables. Del mismo modo pasa en las computadoras, y de una manera mucho más rígida.

En las computadoras actuales se pueden identificar distintos tipos de información, desde datos primitivos, es decir datos atómicos (que están en su forma mínima; en un contexto de funcionamiento de un programa, no son reducibles, por ejemplo, los números enteros), hasta estructuras compuestas de lo anterior. Toda esta abstracción permite modelar el mundo en el que actualmente todo ser humano habita.

2.2.1 Primitivos

Antes de comenzar la clasificación cabe mencionar que en Ruby directamente no existen los tipos de datos primitivos, como se menciona anteriormente en el documento, Ruby es un lenguaje puramente orientado a objetos, lo que indica que todo se concibe como un objeto. Tanto los datos a mencionar como los literales de operadores están representados bajo los conceptos de clases y objetos. Sin embargo, se mantendrá la noción de datos primitivos para comprender a nivel general los tipos de datos dentro de la programación.

- **Enteros**

En lenguaje de código se les conoce como *int*, son un tipo de dato que representa una cantidad numérica entera positiva o negativa. Existe también su versión sin signo, la diferencia radica en la cantidad de memoria usada para representar cada uno.

- **Flotantes**

Abarcan los números con decimales. Se destacan por precisar la magnitud de un número, si es grande o pequeño en comparación con otros.

- **Booleanos**

Son datos con dos posibles valores, verdadero o falso. En programación suelen representarse como 0’s o 1’s, al igual que utilizando las convenciones de ***true*** o ***false***. Los operadores lógicos y de comparación producen resultados de este tipo

de dato, usados mayormente en estructuras de control como el *for, while* o *if*

- **Caracteres**

Un carácter es la forma más básica de representación de las letras y todo lo relacionado a texto y símbolos. Ruby decide ser más general y agrupara todo esto como un solo tipo de dato, *string*.

- **Strings**

Como su nombre en inglés indica (cadenas en español) son cadenas compuestas por caracteres. En los lenguajes de alto nivel, como lo es Ruby, el manejo de este tipo de datos suele ser bastante dinámico y flexible.

2.2.2 Compuestos

Anteriormente se determinaron los datos compuestos como una estructura formada por los datos primitivos conocidos. Si bien es cierto la definición es muy general, por lo que una breve estandarización podría ser el verlos como un conjunto de datos que abstraen aún más la información proporcionada con respecto a su organización y manejo. Este tipo de datos puede ser conocido con el término de TAD o tipos de datos abstractos.

Estos TAD comparten con los otros tipos de datos mencionados el objetivo de representación y manejo de la información, con la diferencia de que estos son a grandes escalas mientras que aquellos que son primitivos son a escalas de unidad.

La existencia de este tipo de datos se debe al hecho de la creciente cantidad de información virtual generada por la raza humana, y con el fin de poder hacer algo se optó por proporcionar una herramienta (de todo el equipo conceptual y por ende material necesario) que fuera útil en el manejo de la misma. Otra de sus razones es el beneficio de velocidad y eficiencia (en algunas) de espacio físico y virtual en el tratamiento de la información.

Tipos de datos abstractos

Algunos de los tipos de datos abstractos más utilizados y conocidos son:

- Arreglos
- Matrices
- Listas
- Pilas y colas
- Tablas hash

- Árboles binarios

Más adelante en el desarrollo del documento se ahondará en el concepto y funcionamiento de algunos de estos tipos de datos.

2.3 Variables

Se identifican como tipos de dato que almacenan información la cual puede estar cambiando. Este concepto permite referenciar algún valor clasificado según los tipos de datos vistos anteriormente.

La particularidad de Ruby de ser un lenguaje puramente orientado a objetos permite tratar todo lo expresado por medio del lenguaje de manera tal que no se tenga que especificar en cada momento el tipo de dato a utilizar, y de aquí proviene su capacidad de ser de tipado dinámico. Por ejemplo, si se comparan la forma de hacer una suma entre dos números entre el lenguaje de Ruby, con otro que no posea la característica anterior como lo puede ser C++, se tendría el siguiente resultado:

Código escrito en C++ **Resultado**

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6
7      int a,b;
8      a = 90;
9      b = 50;
10     int suma = a + b;
11
12     printf("%d + %d = %d\n", a,b,suma);
13
14     return 0;
15
16
17 }
```

```
$ 90 + 50 = 140
```

Código escrito en Ruby Resultado

```
1  a = 90
2  b = 50
3  suma = a + b
4  printf("%d + %d = %d\n",a,b,suma)
```

```
$  90 + 50 = 140
```

Antes que nada, se puede observar la longitud de cada código, por un lado, la solución propuesta en C++ es más grande en comparación con el programa propuesto en Ruby. Por otra parte, el nivel de detalle de instrucciones es mucho más grande en C++, por lo que este se convierte naturalmente en un lenguaje con una sintaxis demasiado rígida, mientras que Ruby tiene la capacidad de poder ejecutar la expresión de la manera más general posible.
Nótese que las variables utilizadas en la operación *suma* en el programa de C++ deben de estar necesariamente indicadas con el tipo de dato que almacenarán (si es entero, char, bool, entre otros), mientras que Ruby permite tomar cualquier expresión (de base) como si fuese una variable, o bueno, un objeto que almacenará un valor que pueda ser cambiado en cualquier momento, diferencia radical con su contraparte, las constantes.

Esta clasificación de los datos permite poder dividir y analizar problemas de una manera mucho más fácil y concreta puesto que todo lo observable se compone a cierto nivel general, de datos, de información. Con ello se podrá desenvolver el lector a lo largo de los capítulos observando los ejemplos planteados, y en los que no cabe duda de que se utilizará toda la información relacionada al respecto.

2.3.1 Clasificación de las variables según su alcance

Cuando se menciona “según el alcance” se hace referencia al espacio existente accesible por una variable, es decir, dependiendo del lugar en donde se cree o se “declare” la variable, se determinará si esta es local o global.

- **Globales:** Variables que pueden ser usadas en todo momento y lugar.
- **Locales:** Variables que pueden ser únicamente utilizadas en su ambiente de declaración como por ejemplo las funciones, estás utilizan variables locales para definir su funcionamiento interno.

Un ejemplo sería el siguiente:

```
1  b = "variable local que solo se puede usar por fuera de la función"
2  def something()
3    a = "variable local que solo se puede usar dentro de la función"
4    return a,b
5
6  end
7
8  puts something()
```

La ejecución del código anterior es:

```
$  Traceback (most recent call last):

   1: from /Users/../LIBROS/LIBRO DE RUBY/CÓDIGO/OTROS
   VARIOS/variables_scope.rb:9:in `<main>'
   /Users/miguellopez/Desktop/LIBROS/LIBRO DE RUBY/CÓDIGO/OTROS
   VARIOS/variables_scope.rb:3:in `something': undefined local variable or
   method `b' for main:Object (NameError)
```

Para entender un poco más lo que sucede con el código, hay que tener claro lo que se intenta hacer. A partir de la definición de una función *something()* y de las expresiones *a* y *b*, una dentro de la función y otra fuera , lo que se quiere intentar es tratar de demostrar el concepto de variable local. Tanto *a* como *b* son variables locales, *a* solo puede ser usada en el ambiente de definición de la función donde ha sido declarada, ósea *something()*, mientras que *b* solo puede ser usada en su ambiente de declaración, es decir por fuera de la función *something()*.

La ejecución indica que se está usando algo de forma indefinida en el método (o función a término general) llamado “b”, lo cual vendría siendo el intento de poder acceder a una variable local por fuera del ambiente que se está trabajando. *b* está definido por fuera de la función, más como se caracteriza por ser local esta no puede ser accedida dentro de la función, lo que resultaría entonces en el error arrojado. Caso similar ocurrirá si se intenta acceder a la expresión *a*.

Ahora bien, para ilustrar el caso de una variable global se puede hacer lo siguiente:

```
1  $b = "variable local que solo se puede usar por fuera de la función"
2
3  def something()
4    a = "variable local que solo se puede usar dentro de la función"
5    return a,$b
6
7  end
```

```
8
9 puts something()
```

La ejecución de lo anterior sería:

```
$ variable local que solo se puede usar dentro de la función
  variable local que solo se puede usar por fuera de la función
```

En el código anterior se tiene un cambio que permite dar grandes diferencias en cuanto al tipo de expresiones que se han mencionado. Nótese que la variable *b* esta vez viene expresada por el carácter especial *$.* Esta nueva expresión indica que una variable puede ser usada en cualquier ambiente, ya sea dentro de la declaración de alguna función o por fuera de la misma. A la par se tiene también que para poder acceder a ella se debe de usar tal cual la forma como se declaró incluyendo el carácter mencionado. Gracias a todo esto se pueden ejecutar las operaciones dadas.

2.4 Constantes

Al igual que las variables, son expresiones que almacenan valores con información constante, que no puede ser cambiada en ningún momento. En un programa pueden definirse matemáticamente valores constantes como el valor pi (π) o el número de Euler (*e*).

También se dividen en dos grupos, constantes globales y constantes locales, que al igual que en las variables, la primera será de uso general para todo momento y espacio, mientras que la segunda sólo podrá utilizarse en su ambiente declarativo (donde se crea).

2.5 Imprimir y obtener información por pantalla

La información que es utilizada por un programa puede ser obtenida únicamente por medio del ingreso de datos como por ejemplo el uso de la terminal o de interfaces gráficas, usado en programas como aquellos conectados a una base de datos, archivos de texto o los que se disponen en formato CSV (Comma Separated Values). Por esta razón es primordial conocer los métodos de obtención de información, y en este caso, su estructura en el lenguaje de Ruby.

Cada programa tiene sus particularidades al respecto, en el caso de Ruby se tiene una simpleza deslumbrante para el manejo del control de flujo de los datos, la cual radica en el uso de funciones que gestionan la obtención y almacenamiento de información como su manejo para mostrarse al usuario. Dichas funciones son:

- **Puts:** Poner en inglés, es la función que permite mostrar datos en pantalla principalmente dentro de una terminal.
- **Gets:** Obtiene la información introducida (también por medio de la terminal). Una particularidad es que esta información para ser almacenada se debe usar alguna expresión como lo son las variables o incluso las constantes.

Ahora bien, un ejemplo de lo anterior sería:

```
1  puts "Cuál es su edad? -> "
2  age = gets
3  puts "Su edad es #{age}"
```

La salida del código anterior:

```
$  ¿Cual es su edad? ->
   13
   Su edad es 13
```

Hay una característica particular en el código anterior, nótese que en el mensaje “cuál es su edad” la idea de la flecha es indicar que justo después de ella debe ir el mensaje, más sin embargo en la ejecución no se muestra de esa forma. Esto es debido al uso de la herramienta *puts,* la cual siempre muestra mensajes en pantalla dejando al final un salto de línea. Por suerte Ruby soporta el uso de otras generalidades como la expresión *print()* que se comporta de manera similar y descrita, con la diferencia que este último no deja saltos de línea. Con el siguiente ejemplo se puede obtener lo deseado:

```
1  Print "Cual es su edad? -> "
2  age = gets
3  puts "Su edad es #{age}"
```

La ejecución:

```
$  Cual es su edad? -> 13
   Su edad es 13
```

Otro aspecto a tener en cuenta es el detalle mostrado en el mensaje de la línea 3 de los dos códigos anteriores. Como se puede observar, se muestra un pequeño texto en donde con ayuda de la expresión *# {…}* se pueden mostrar variables de una manera más eficiente, ordenada y corta; es por esa razón que dentro de las llaves precedidas por el numeral se escribe la variable o expresión que almacén los datos guardados por *gets*. Por otra parte, la impresión de variables y múltiples datos puede ser presentado de la manera convencional:

```
1  Print "Cual es su edad? -> "
2  age = gets
3  puts "Su edad es " + age + " años"
```

Se observa que gracias a la operación de suma aplicado para textos o cadenas de caracteres se puede manejar múltiples de un mensaje, como se muestra en la línea

El resultado sería:

```
$  Cual es su edad? -> 13
   Su edad es 13
   años
```

Surge otra situación y es el cambio de línea luego de mostrar la edad ingresada. Esto es debido a la función *gets* que de modo automático asigna un salto de línea en la ejecución luego de que el valor fue ingresado. Si se desea esto puede solucionarse utilizando algunos de los métodos diseñados para *gets* (recordar que todo en Ruby es un objeto):

```
1  Print "Cual es su edad? -> "
2  age = gets.chomp
3  print("Su edad es " + age + " años")
```

Su ejecución:

```
$  Cual es su edad? -> 13
   Su edad es 13 años
```

Nota: La sintaxis de Ruby no es rígida en cuanto al uso de paréntesis y otros caracteres que definen una expresión. Por ello se tiene la libertad de llamar métodos de un objeto con o sin paréntesis, aunque estos dependen de si el método a usar tiene o no parámetros definidos, como de si se necesitan mandar o no parámetros extras, entre otras cosas.

2.6 Ejemplos de lo visto en el capítulo

En esta sección se explicarán algunos códigos que agrupan toda la información recopilada en el capítulo 2 de este documento. En ellas se podrá observar el uso de los operadores aritméticos en forma de expresiones matemáticas, variables (en su caso locales, las globales podrán ser mejor entendidas en el uso de funciones) que almacenen datos ingresados por el usuario, entre otras características.

Ejemplo #1

Números enteros y flotantes

Se presenta a continuación el siguiente código:

```
1  #Enteros y flotantes
2  puts "Ingrese dos numeros a y b "
3  number_a = gets.chomp.to_i
4  number_b = gets.chomp.to_i
5
6  print("Los números ingresados son \n\t
7       #{number_a} es de tipo #{number_a.class} \n\t
8       #{number_b} es de tipo #{number_b.class}\n\n")
9
10 puts "Algunas operaciones entre ellos son:"
11 puts "Suma -> a + b = #{number_a + number_b}"
12 puts "Resta -> a - b = #{number_a - number_b}"
13 puts "Multiplicación -> a * b = #{number_a * number_b}"
14 puts "División -> a / b = #{number_a / number_b}"
15
16 puts "\n\nde igual forma las operaciones entre flotantes: "
17
18 number_a = gets.chomp.to_f
19 number_b = gets.chomp.to_f
20 print("Los números ingresados son \n\t
21      #{number_a} es de tipo #{number_a.class} \n\t
22      #{number_b} es de tipo #{number_b.class}\n\n")
23 puts "Suma -> a + b = #{number_a + number_b}"
24 puts "Resta -> a - b = #{number_a - number_b}"
25 puts "Multiplicación -> a * b  = #{number_a * number_b}"
26 puts "División -> a / b = #{number_a / number_b}"
```

En el código anterior se utilizan las funciones necesarias para la obtención e impresión de datos por pantalla (*puts, print, gets*). Este funciona de tal manera que se pueda obtener dos números, con los cuales se podría realizar operaciones aritméticas básicas como la suma, resta, multiplicación y división. Por otra parte, hay ciertas características como el uso de la estructura *#{…}* o el manejo de *castings* (de manera general es un término que se refiere a cambio o paso de un tipo de dato a otro, por ejemplo, pasar un dato de tipo *string* a tipo entero) para poder convertir la información ingresada por el usuario en datos manejables por las operaciones descritas.

Una prueba de lo que puede realizar el código anterior es:

$	Ingrese dos numeros a y b
	1
	2
	Los números ingresados son
	1 es de tipo Integer
	2 es de tipo Integer
	Algunas operaciones entre ellos son:
	Suma -> a + b = 3
	Resta -> a - b = -1
	Multiplicación -> a * b = 2
	División -> a / b = 0
	de igual forma las operaciones entre flotantes:
	4
	2
	Los números ingresados son
	4.0 es de tipo Float
	2.0 es de tipo Float
	Suma -> a + b = 6.0
	Resta -> a - b = 2.0
	Multiplicación -> a * b = 8.0
	División -> a / b = 2.0

Ejemplo #2

Manejo de expresiones matemáticas por medio de la jerarquía de operandos

```
1  # evaluar expresiones matemáticas y jerarquía de operandos
2  puts "Usted va a evaluar la siguiente expresión a partir del reemplazo con datos propocionados por usted:"
3  puts "expresión -> (number_a + number_c) * number_b / number_c + ((number_c-number_a)*(number_d/(number_a)))"
4  puts "A continuación ingrese un valor para cada una de las variables que componen la expresión anterior"
5  print("number_a: ")
6  number_a = gets.chomp.to_i
7  print("number_b: ")
8  number_b = gets.chomp.to_i
9  print("number_c: ")
10 number_c = gets.chomp.to_i
11 print("number_d: ")
12 number_d = gets.chomp.to_i
13
14 polinomio_arbitrario = (number_a + number_c) * number_b / number_c + ((number_c-number_a)*(number_d/number_a))
15
16 puts "La respuesta a la expresión anterior con los valores suministrados es " + polinomio_arbitrario.to_s()
```

El código anterior permite evaluar la expresión *polinomio_arbitrario* al ingresar datos para los elementos que lo conforman, estos son *number_a*, *number_b*, *number_c*, *number_d***.** Estos elementos son variables locales las cuales almacenan el valor retorna por la función *gets* en compañía de *chomp* y un *casting* (cambio de un tipo de dato a otro) hacia un dato de tipo entero.

Analice un momento la expresión de *polinomio_arbitrario*, observe que se compone únicamente de paréntesis como indicadores de jerarquía. Debe de existir un número igual de parejas de paréntesis para abrir o cerrar, de lo contrario se arrojaría un error al momento de la ejecución. Para verificar que el concepto de jerarquía de operadores se entiende se propone calcular el valor que calcularía *polinomio_arbitrario* si los valores de sus elementos fuesen:

number_a = 1
number_b = 2
number_c = *3*
number_d = *4*

Nota: al tener una división entre números enteros, Ruby de modo automático aproxima el resultado a la parte entera que lo compone si no se especifica el tipo de resultado obtenido, es decir, para el caso que se tiene del ejemplo donde 32 es dividido por 3, el resultado debería de ser 10.66666667, sin embargo, al no especificarse que será un dato de tipo flotante el valor que toma la variable donde esta operación se almacena es 10.

Ahora bien, si se ejecuta el código, el resultado de *polinomio_arbitrario* debe de ser igual:

$	Usted va a evaluar la siguiente expresión a partir del reemplazo con datos porpocionados por usted:
	expresión -> (number_a + number_c) * number_b / number_c + ((number_c-number_a)*(number_d/(number_a)))
	A continuación ingrese un valor para cda una de las variables que componen la expresión anterior
	number_a: 1
	number_b: 2
	number_c: 3
	number_d: 4
	La respuesta a la expresión anterior con los valores suministrados es 10

Como se puede observar, el análisis escrito de la expresión *polinomio_arbitrario* es correcto comparado con los resultados que arroja el código sugerido.

2.7 Ejercicios propuestos

Se proponen los siguientes ejemplos:

1. Crear un programa que permita calcular el volumen de un cilindro, una esfera y un cono. Las ecuaciones de las respectivas figuras son:

$$\text{Cilindro} = \pi r^2 h$$

$$\text{Esfera} = \frac{4}{3}\pi r^3$$

$$\text{Cono} = \pi r^2 h$$

   Al ejecutar el programa, este deberá estar en capacidad de recibir los datos necesarios para el cálculo de todos los volúmenes y mostrar los resultados en pantalla.

2. Escribir un código que reciba un número entero correspondiente a la edad de un usuario y que retorne el número de Meses, días y horas que lleva vivo.

3. Crear un programa que partiendo del ingreso de números enteros y flotantes permita calcular polinomios de segundo y tercer grado.

   Se recomienda que el lector consulte temas relacionados con el uso de librerías matemáticas, como por ejemplo funciones que calculen la raíz cuadrada de un número natural.

4. En un teatro un cliente paga $20 por la entrada y cada función le cuesta al teatro $300. Por cada cliente que ingresa al teatro debe pagar un costo de $2 por aseo adicional (es decir, $20 de la entrada más $2 del aseo). Desarrollar un programa que reciba el número de clientes de una función y devuelva el valor de las ganancias obtenidas por el teatro entendiendo que para que el teatro obtenga ganancias, el número de clientes que se deben de alcanzar es más de 15 (es decir, *cliente >= 15*).

5. En una tienda de variedades se ofrecen descuentos en cada categoría. Las categorías son: juguetes para niños con 10%, cuadros de arte con el 5%, productos de maquillaje con el 7% y ropa con el 9%. Un cliente desea llevar $100.000 en juguetes para niños, $50.000 en cuadros de arte, $35.000 en productos de maquillaje y $100.000 en ropa. Se necesita calcular el descuento total por cada tipo de producto, posteriormente entregar el valor total a pagar con descuento y sin descuento.

6. Descomponer un número de 5 cifras en unidades (1), decenas (10), centenas (100), milésimas (1000) y diez milésimas (10000) y decir la cantidad correspondiente, ejemplo: el número 631 tiene 1 unidad, 3 decenas y 6 centésimas.

7. Del ejercicio anterior, descomponer un número e invertirlo, por ejemplo, 582 pasaría a ser 285.

3 Estructuras de control y flujo

Las estructuras de control son aquellos modelos que permiten modificar el flujo (continuidad o camino) de información ya sea por medio de pausas para realizar operaciones, como determinar condiciones, entre otras. Basada en la programación estructurada o por módulos, Ruby permite estructuras de repetición y estructuras condicionales, las cuales tienen como característica común de trabajar de acuerdo a una condición generada por la obtención de un valor a través de uno o más procesos.

Cada estructura es similar en la mayor parte de lenguajes de programación con la diferencia que existen múltiples versiones con mejoras, como la sintaxis y posiblemente comandos especiales.

Para el caso de estudio de este documento las estructuras se definen según las siguientes secciones.

3.1 Condicionales

También llamadas de selección, son estructuras que proporcionan un conjunto de acciones siempre y cuando se cumpla una condición. Algunos modelos son:

3.1.1 Estructura de condición if

Gracias a la característica de Ruby ya mencionada, su sintaxis se vuelve mucho más amigable, por lo que la estructura de los condicionales se representaría como:

```
If( <condiciones> )
        <Bloque de instrucciones #1>
else
        <Bloque de instrucciones #2>
```

El campo de condiciones puede ser reemplazado con expresiones que retornan un valor *booleano* ya sea verdadero o falso lo que repercutirá en que se ejecute el *bloque de instrucciones #1* o el *bloque de instrucciones #2 según sea el caso respectivamente.* Por otra parte, estas expresiones pueden componerse a partir del uso de operadores aritméticos y/o de operadores lógicos.

Este modelo permite el anidamiento de estructuras, es decir, la creación de nuevas estructuras dentro de una estructura inicial, así como el uso de múltiples condiciones con su versión *if-elsif-else.*

3.1.1.1 Estructura de condición if-elsif-else

De manera similar cumple el mismo papel que la estructura base de una condición *if,* con la diferencia de que, en lugar de tener únicamente dos opciones resultantes de una sola comparación, esta permite evaluar múltiples condiciones en una sola estructura y proveer múltiples opciones resultantes. Su representación es:

```
If( <condiciones> )
        <Bloque de instrucciones #1>
elsif
        <Bloque de instrucciones #2>
.
.
.
else
        <Bloque de instrucciones #n>
```

3.1.2 Condicional case-when

Al igual que el *if*, es un condicional que ejecuta un conjunto de operaciones dependiendo de un valor resultante. Se compone de la siguiente manera:

```
case (<variable>)

   when <Valor 1 variable> {<bloque de instrucciones #1>}
   when <Valor 2 variable> {<bloque de instrucciones #2>}
   .
   .
   .
   when <Valor n variable> {<bloque de instrucciones n>}
   else {<bloque de instrucciones>}
end
```

De forma general el conjunto de pasos que se ejecutan es:

- *El campo variable* almacena un valor el cual servirá para determinar un cierto conjunto de instrucciones a ejecutar.
- Se recorren todos los *when*, buscando aquel que sea igual al valor de *variable*.
- Si se encuentra aquel que cumpla la condición se encuentra se ejecuta el bloque de instrucciones.
- Caso contrario se ejecuta el *else* (si se propone) con su bloque de instrucciones. Este *else* será un caso *default* o caso general que servirá para indicar que no hay valores en la estructura *case-when* que coincidan con el valor de *variable.*
- Se termina la ejecución del *case-when* al llegar al *end*.

3.2 Iteradores

Los iteradores o estructuras de repetición componen un conjunto de operaciones que se ejecutan un determinado número de veces, esto hasta que se cumpla o se mantenga una condición. Algunos de los modelos más utilizados son el *for* y el *while.* Estos pueden variar su presentación, algunas de ellas son:

3.2.1 For

for <variable> in [variable o tope] do
<bloque de operaciones>
end

<*variable*> es un tipo de contador encargado de ejecutar las operaciones en <*bloque de operaciones*> el número de veces determinado por [*variable o tope*]

3.2.2 While

while <condición> do
<bloque de operaciones>
end

De manera similar al *for*, el *while* se ejecuta un cierto número de veces según lo determine una condición. Esta condición puede tener un contador como ocurre con un ciclo *for* utilizando el valor de verdad de las expresiones lógicas, o simplemente no tenerlo.

3.2.3 While modificado

<bloque de instrucciones> while <condición>

También

begin
<bloque de instrucciones>
end while <condición>

3.2.4 Until

until <condición> do
<bloque de instrucciones>
end

3.3 Ejemplos de uso

En esta sección se observarán algunos ejemplos sobre el funcionamiento de las estructuras iterativas y de condición, tanto su desarrollo general como algunas de sus variantes.

3.3.1 Ejemplos que involucren condiciones

- Crear un programa con los datos que el usuario ingresa por teclado y posteriormente retorna una característica del producto con su respectivo pecio.

Código solución

```
1  print("ESTA ES LA LISTA DE PRODUCTOS TECNOLÓGICOS DISPONIBLES:\n\t\t
2      CELULARES:\n\t\t
3      1) Iphone 12 pro MAX
4      2) Xiaomi RedMi Note 10 Pro
5      \n
6      \t\t
7      TARJETAS GRÁFICAS:\n\t\t
8      3) RTX 3080 ti
9      4) RTX 3060
10     \n
11     \t\t
12     BEBIDAS ENERGETICAS:\n\t\t
13     5) Speed Max 350 ml
14     6) MONSTER 600 ml
15 ")
16 opcion = gets.chomp.to_i
17
18 case opcion
19
```

```
when 1,2
  if opcion == 1
    puts "Iphone 12 pro MAX -> $4.500.000\n
        Procesador y megapixeles de camara trasera -> #{"Apple A14 Bionic"} - #{"12 MegaPixeles"}"
  elsif opcion == 2
    puts "Xiaomi RedMi Note 10 Pro -> $2.350.000\n
        Procesador y megapixeles de camara trasera -> #{"SnapDragon 720"} - #{"34 MegaPixeles"}"
  end
when 3,4
  if opcion == 3
    puts "RTX 3080 ti -> $5.250.000\n
        VRAM y consumo/potencia (Watts) -> #{"16 GB"} - #{"350 Watts"} "
  elsif opcion == 4
    puts "RTX 3060 -> $3.150.000\n
        VRAM y consumo/potencia (Watts) -> #{"8 GB"} - #{"170 Watts"}"
  end
when 5,6
  if opcion == 5
    puts "Speed Max 350 ml -> $3.000\n
        Azucar y cafeína -> #{"17 gramos cada porción (2 porciones)"} - #{"500 mg"}"
  elsif opcion == 6
    puts "MONSTER 600 ml -> $6.500\n
        Azucar y cafeína -> #{"34 gramos cada porción (2 y media porciones)"} - #{"1000 mg"}"
```

```
43    end
44
45  end
```

El funcionamiento general del código describe un conjunto de opciones usando la estructura *if-elsif-else* y *case-when*. A partir del ingreso de un número comprendido entre 1 y 6 el cual se almacena en la variable *opcion* (y que a la vez maneja las funciones para recibir datos por teclado) este se evalúa en la estructura *case-when* y según el caso se ejecutan diferentes instrucciones, por ejemplo, al ingresar el número 5 la respuesta será las características del producto *speed max*, mientras que si se ingresa el número 3 la respuesta será las características del producto *RTX 3080 ti*.

Algunos ejemplos de ejecución son:

Con la opción No. 1:

```
$  ESTA ES LA LISTA DE PRODUCTOS TECNOLÓGICOS DISPONIBLES:

      CELULARES:

      1) Iphone 12 pro MAX
      2) Xiaomi RedMi Note 10 Pro




      TARJETAS GRÁFICAS:

      3) RTX 3080 ti
      4) RTX 3060




      BEBIDAS ENERGETICAS:

      5) Speed Max 350 ml
      6) MONSTER 600 ml
  1
  Iphone 12 pro MAX -> $4.500.000

        Procesador y megapíxeles de cámara trasera -> Apple A14 Bionic - 12 MegaPíxeles
```

Cabe resaltar que lo que causa los grandes espacios en blanco entre cada mensaje producido de la ejecución es gracias al uso de las funciones *puts* y los caracteres especiales '*n*' el cual indica que la siguiente impresión se haga en una nueva línea. Existen otros tipos de caracteres especiales como '*t*' para tabulación, '\s' o '*space*' para generar un nuevo espacio (el espacio usado con la barra espaciadora del teclado), entre otros.

- Crear un programa que indique (usando solo condicionales), cual es el menor de 3 números

Código solución

```
1  puts "Ingrese tres numeros distintos: "
2  numero_a = gets.chomp.to_i
3  numero_b = gets.chomp.to_i
4  numero_c = gets.chomp.to_i
5  print "Elija una opción:"
6  opcion = gets.chomp.to_i
7
8  case opcion
9  when 1
10   puts "Método #1"
11
11   if numero_a < numero_b
12
12     if numero_a < numero_c
13
13       puts "El menor numero de #{numero_a} #{numero_b} #{numero_c} es el numero #{numero_a}"
14     elsif numero_b < numero_c
15
15       puts "El menor numero de #{numero_a} #{numero_b} #{numero_c} es el numero #{numero_b}"
16     else
17
17       puts "El menor numero de #{numero_a} #{numero_b} #{numero_c} es el numero #{numero_c}"
18     end
19
19   else
20
20     if numero_b < numero_c
21
21       puts "El menor numero de #{numero_a} #{numero_b} #{numero_c} es el numero #{numero_b}"
```

```
22    else
23      puts "El menor numero de #{numero_a} #{numero_b} #{numero_c} es el numero #{numero_c}"
24    end
25  end
26  when 2
27    puts "Método #2"
28    #método #2
29
30    if numero_a < numero_b and numero_a < numero_c
31      puts "El menor numero de #{numero_a} #{numero_b} #{numero_c} es el numero #{numero_a}"
32    elsif numero_b < numero_a and numero_b < numero_c
33      puts "El menor numero de #{numero_a} #{numero_b} #{numero_c} es el numero #{numero_b}"
34    else
35      puts "El menor numero de #{numero_a} #{numero_b} #{numero_c} es el numero #{numero_c}"
36    end
37  end
```

El código anterior resuelve la situación descrita por dos métodos diferentes, cada uno se emplea según los siguientes análisis:

- El método #1 se basó en el hecho de que al tener tres números (a,b,c), independiente del hecho que estén o no ordenados, se deben de hacer las comparaciones empezando por el primero y avanzando por cada uno si no se cumple el requisito de que uno tiene que ser menor que los restantes.

  En el código, más específicamente la forma en la que se estructura la condición indica que al iniciar con cada comparación hay la posibilidad de que se eviten números en comparaciones futuras, es decir, si el proceso de comparación empieza por el número *a* comparándolo con *b* y *c*, y dado el caso que *a* no sea menor que alguno de los dos anteriores, se puede concluir

que la respuesta esté comparando únicamente *b* y *c*, que serían los números más pequeños después de *a*. En ese orden de ideas, continuaría el proceso con el número *b* comparándose con *c*, repitiendo el patrón anterior de descarte.

- El método #2 se basa en la idea de que un número es menor a otro si y sólo ese número es menor que todos los demás, por ejemplo, de los tres números (*a, b, c*) mencionados, *a* sería el menor si es menor que *b* y *c*, y así sucesivamente sucedería con *b* y *c*.

Cada una de las condiciones que rigen los métodos explicados anteriormente son descritas por medios de condicionales *if-elsif-else* y las estructuras *case-when* .

Al ejecutar el código y probarlo con la tripleta de números 1, 2, 3 en algunas de sus combinaciones se obtiene que:

- Usando el método No. 1:

```
$ 1
  2
  3
  Elija una opción:1
  Método #1
  El menor numero de 1 2 3 es el numero 1
```

- Usando el método No. 2:

```
$ 1
  2
  3
  Eliga una opción:2
  Método #2
  El menor número de 1 2 3 es el numero 1
```

3.3.2 Ejemplo que involucren iteraciones

- Calcular la serie de Fibonacci a partir de un límite especificado

Código solución

```
1 puts "Ingrese un numero limite para calcular la serie de números Fibonacci"
2 limite = gets.chomp.to_i
3 n_menos_1 , n , suma = 0,1,1
```

```
4
5  puts "La seríe de fibonacci en un rango de [0....limite] es: "
6  for iteracion in 0..limite do
7    print(suma, " ")
8    suma = n_menos_1 + n
9    n_menos_1 = n
10   n = suma
11 end
```

La famosa seríe de Fibonnaci, la cual inicia como *0,1,1,2,3,5,8,13,21,34, 55, [n = (n-2) + (n-1)]* también puede ser representada por medio de una estructura de iteración *for.* El razonamiento para poder ejecutar lo solicitado es el siguiente:

- A partir de la fórmula que describe la serie y dada su secuencia puede ser infinita, se escoge por determinar un límite de cálculo, es decir, escoger el número de dígitos pertenecientes a la serie que serán calculados.
- Con un límite definido, utilizando una estructura de iteración se puede determinar la fórmula iterando desde el caso base, el número cero hasta el dígito limite. El cálculo de un número perteneciente a la serie se logra mediante la definición de *n_menos_1, n* y *suma*, variables las cuales representarán *n-2, n-1* y (*(n-1) +(n-2)*) respectivamente. Cabe mencionar que los primeros valores presentes en la serie al ser casos base pueden ser proporcionados como valores iniciales en la declaración de cada una de las variables mencionadas.
- Con todo lo definido hasta este punto se puede crear el programa, en este caso se pide ingresar la información de un límite para el cálculo de la serie, posteriormente luego de la declaración de las variables se crea una estructura *for* que itera desde un rango de cero hasta el límite ingresado. En este se define y se reasignan valores en la operación de la serie de Fibonacci, los cuales a su vez serán presentados en la pantalla de la terminal.

Ejecutando lo anterior se obtienen los siguientes resultados:

- Usando como límite de cálculo el número 100:

La serié de fibonacci en un rango de [0....limite] es:

***Orden de lectura*

```
$  1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
   17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269
   2178309 3524578 5702887 9227465 14930352 24157817 39088169
   63245986 102334155 165580141 267914296 433494437 701408733
   1134903170 1836311903 2971215073 4807526976 7778742049
```

	12586269025 20365011074 32951280099 53316291173 86267571272 139583862445 225851433717 365435296162 591286729879 956722026041 1548008755920 2504730781961 4052739537881 6557470319842 10610209857723 17167680177565 27777890035288 44945570212853 72723460248141 117669030460994 190392490709135 308061521170129 498454011879264 806515533049393 1304969544928657 2111485077978050 3416454622906707 5527939700884757 8944394323791464 14472334024676221 23416728348467685 37889062373143906 61305790721611591 99194853094755497 160500643816367088 259695496911122585 420196140727489673 679891637638612258 1100087778366101931 1779979416004714189 2880067194370816120 4660046610375530309 7540113804746346429 12200160415121876738 19740274219868223167 31940434634990099905 51680708854858323072 83621143489848422977 135301852344706746049 218922995834555169026 354224848179261915075 573147844013817084101

- Elaborar un programa que permita imprimir en pantalla las siguientes figuras:

```
          *
        * * *
      * * * * *
    * * * * * * *
  * * * * * * * * *
* * * * * * * * * * *
```

```
* * * * * * * * * * * * *
  * * * * * * * * * * *
    * * * * * * * * *
      * * * * * * *
        * * * * *
          * *
```

Código solución

Forma #1

```
#     *
#    ***
#   *****
#  *******
# *********

print "Digite la cantidad de nivéles que desea visualizar en la piramide: "
niveles = gets.chomp.to_i
espacios  = niveles
numero_asteriscos = 0


puts ""
for iteracion in 0..niveles do
  print(" "*espacios)
  numero_asteriscos = (2*iteracion) + 1
  for asteriscos_iteracion in 1..numero_asteriscos
    print("*")
  end
  puts ""
  espacios -= 1
end
```

Forma #2

```
# ******
#  ****
#   **    Números pares por cada nivel
#
print "Digite la cantidad de nivéles que desea visualizar en la piramide: "
niveles = gets.to_i
espacios  = 0
puts ""
for iteracion in (niveles).downto(0) do
  print(" "*espacios)
```

```
10
11    numero_asteriscos = (2*iteracion)
12    for asteriscos_iteracion in 0..(numero_asteriscos-1)
13      print("*")
14    end
15    puts ""
16    espacios += 1
17  end
```

Los dos casos anteriores manejan un análisis similar, es decir, para poder representar dichas pirámides se debe de tener en cuenta los siguientes factores:

- El número de niveles a representar.
- El número de espacios que hay para poder imprimir cada nivel.
- El número de asteriscos impresos por cada nivel.
- Determinar el momento en el cual la impresión de asteriscos debe detenerse para poder dar la figura indicada.
- Entre otros.

Ejecutando el código se logra obtener la figura solicitada:

Figura #1

Ilustración 3. Ejecución del programa que imprime una pirámide invertida

Figura #2

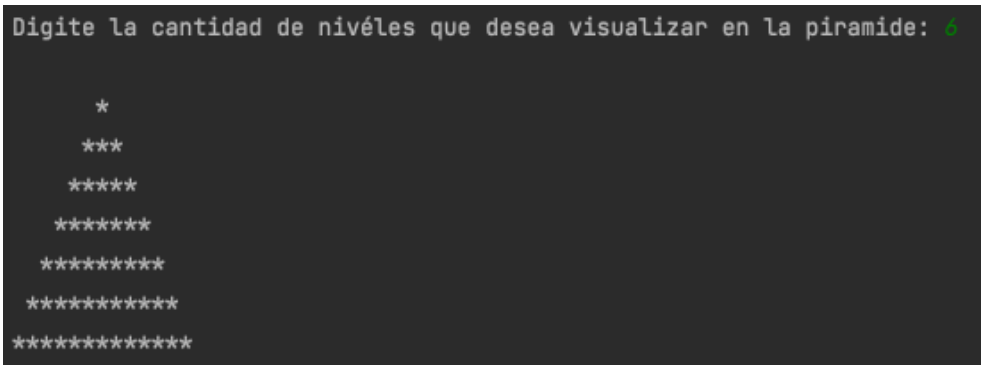


Ilustración 4. Ejecución del programa que imprime una pirámide

Para entender un poco mejor lo que se está realizando se explicará el primer ejemplo de forma detallada, y el segundo quedará a disposición del lector.

Caso #1

La situación que se pide analizar propone la formación de una pirámide invertida usando asteriscos.

Nótese que en la figura #1 las filas se completan de forma descendente y consecutiva hasta llegar a un tope determinado por el usuario. También se puede observar que el número de asteriscos con los que se completa cada fila es par de dos en dos, siendo un caso de ejemplo con un límite de filas igual a seis, la primera fila estaría con doce asteriscos, la segunda con diez, la tercera con ocho, y así sucesivamente hasta llegar al mínimo posible que en este caso sería dos.

Para crear la figura se deben conocer las relaciones existentes y saber cómo debe comportarse el programa en múltiples situaciones. Lo primero es que se tiene que conocer el límite de la pirámide, en otras palabras, la altura o número de filas que esta tendrá. Para ello se pueden utilizar variables que almacenen dicho valor (y el cual deberá ser suministrado por teclado según el ejercicio). Para el caso del código propuesto se usará el nombre de *niveles*.

Otra de las relaciones para tener en cuenta, y que está mencionada anteriormente en el texto, es la impresión del número correcto de espacios por fila. Se puede observar que a medida que el número de asteriscos impresos por fila disminuye, el número de espacios aumenta; dicho comportamiento es de mucha utilidad al momento de desarrollar el componente principal. Con esto en mente se sabe que espacios al ser un número cambiante, también puede ser almacenado por una variable, que para el código mencionado será *espacios*.

Ahora bien, en este momento se puede hacer un gran uso de las estructuras de iteración puesto que se debe de repetir un proceso un número determinado de veces, que es la impresión de espacios y asteriscos. Para ello se utilizará la

estructura *for*. En el código se crea lo siguiente:

```
for iteracion in (niveles).downto(0) do
        …
End
```

Hay una particularidad con lo anterior, y es la forma en cómo se indica que el número de iteraciones irá decreciendo. Se escogió la manera decremental debido a que es el comportamiento con que se deben imprimir los asteriscos en pantalla y, por ende, teniendo en cuenta la forma de definir la estructura *for* , *iteracion* sería la *variable* que iterara de forma decreciente en el rango determinado, ayudando en el proceso de impresión.

Continuando con la característica de cómo se define una estructura *for* que va en decremento, nótese que se usa un función llamada *downto()* a partir del límite almacenado en *niveles*. Primero, esta notación se debe a la característica principal de Ruby, por ser un lenguaje completamente orientado a objetos, este será un tema de discusión en capítulos posteriores. Segundo, si se intenta leer y comprender la línea que utiliza la función anterior, se podrá observar que ello indica una disminución desde *niveles* hasta cero, un proceso de *decremento* que es justamente lo que se desea. La forma definida allí es una de las múltiples que existen de hacer una estructura iterativa en *decremento*.

Estando en el interior de la estructura *for* se tienen las siguientes operaciones:

```
print("  "*espacios)
numero_asteriscos  =  (2*iteracion)
```

La primera es la encargada de mostrar el número de espacios necesarios. Recuerde que *espacios* se inició en cero, esto es debido a que según la forma de la pirámide, al desarrollar la primera fila esta se hace sin ningún espacio, además la función *print()* permite operaciones entre cadenas de texto o *strings* y números, como lo es *“ ” * espacios*, indicando que la cadena vacía *“ ”* se debe multiplicar por el número de *espacios* asignado en esa iteración. La segunda operación calcula el número de asteriscos a imprimir.

Posteriormente se decide crear otra estructura *for* ; como ya se ha mencionado, se necesita imprimir una cierta cantidad de asteriscos por cada fila hasta un límite asignado. Este proceso es a la vez una conjunto de iteración o repetición de tareas, por lo que, así como se utiliza una estructura de iteración que contiene todas las operaciones para crear la pirámide, la impresión de los asteriscos también debe de hacer algo similar.

Luego de pasar por todo lo mencionado hasta este momento, se debe de tener en cuenta el comportamiento de los espacios con respecto al número de asteriscos por cada fila. Si uno disminuye el otro aumenta, siendo lo último referente a la variable

espacios. En el código se especifica el incremento con *espacios* += *1*.

Por último, en ciertas líneas se implementa la impresión de una cadena vacía en una nueva línea con el fin de pasar a la siguiente fila por cada iteración.

En este punto ya se ha terminado el seguimiento y análisis del código anterior, concluyendo un programa que arroje los resultados esperados.

- Dibujar la letra A con asteriscos.

3.3.3 Usando iteraciones y condicionales

- Un programa que se ejecute “infinitamente” lanzando operaciones matemáticas sencillas que el usuario pueda responder. Si este se equivoca en la respuesta el programa se debe seguir ejecutando, todo esto hasta recibir una respuesta correcta.
  Las operaciones serán suma, resta, multiplicación, división, todas entre números aleatorios.

Código solución

```
1  #Herramientas a usar
2  operaciones = [" + "," - "," * "," / "]
3
4  while true do
5    numero_a = rand(0..100).to_s
6    numero_b = rand(0..100).to_s
7    operacion = rand(operaciones.length)
8    calculo = numero_a + operaciones[operacion] + numero_b
9    puts calculo
10   print "Su respuesta -> "
11   respuesta = gets.chomp.to_i
12
13   while respuesta == 0
14     respuesta = gets.chomp.to_i
15   end
16
     case operaciones[operacion]
```

```
17
18    when " + "
19      if numero_a.to_i + numero_b.to_i == respuesta
20        print "RESPUESTA CORRECTA!!!\n"
21        break
22      else
23        print "RESPUESTA INCORRECTA, SIGUE INTENTANDO\n"
24      end
25    when " - "
26      if numero_a.to_i - numero_b.to_i == respuesta
27        print "RESPUESTA CORRECTA!!!\n"
28        break
29      else
30        print "RESPUESTA INCORRECTA, SIGUE INTENTANDO\n"
31      end
32    when " * "
33      if numero_a.to_i * numero_b.to_i == respuesta
34        print "RESPUESTA CORRECTA!!!\n"
35        break
36      else
37        print "RESPUESTA INCORRECTA, SIGUE INTENTANDO\n"
38      end
39    when " / "
```

```
40        if numero_a.to_i / numero_b.to_i == respuesta
41          print "RESPUESTA CORRECTA!!!\n"
42          break
43        else
44          print "RESPUESTA INCORRECTA, SIGUE INTENTANDO\n"
45        end
46      end
47    end
```

El código anterior utiliza una iteración "*infinita*" (*while true*) para poder simular el bombardeo constante de operaciones al usuario hasta el punto en donde se acierte con el resultado de alguna de ellas. El programa utiliza una *lista* (estructura de datos) que almacena los caracteres de los operadores aritméticos elementales (suma, resta, multiplicación, división) así como las funciones *rand()* que retornan valores aleatorios referentes al tipo de operación a utilizar y los dos dígitos o números que conformarán la operación aritmética. Cuando se computa la operación y se proporciona al usuario, el computador recibe la respuesta y la almacena en *respuesta*, variable que posteriormente será comparada con el resultado esperado de la operación computada por medio de la estructura *case-when.* Se mostrará un mensaje de "RESPUESTA CORRECTA!!!\n" o "RESPUESTA INCORRECTA, SIGUE INTENTANDO\n" según el valor retornado de la comparación anterior.

Ejecutando el código anterior se obtiene lo siguiente:

```
$ 40 * 79
  Su respuesta -> 189
  RESPUESTA INCORRECTA, SIGUE INTENTANDO
  84 * 96
  Su respuesta -> 3246
  RESPUESTA INCORRECTA, SIGUE INTENTANDO
  79 + 94
  Su respuesta -> 345
  RESPUESTA INCORRECTA, SIGUE INTENTANDO
  96 * 7
  Su respuesta ->

  35
  RESPUESTA INCORRECTA, SIGUE INTENTANDO
```

	77 / 13
	Su respuesta -> 5
	RESPUESTA CORRECTA!!!

Un último aspecto para detallar es que en la operación de *96 * 7* se introdujeron dos espacios vacíos a causa de presionar la tecla *enter* dos veces. En otras situaciones el código hubiese dejado de ejecutarse puesto que en este proceso toda entrada de información es válida (el enter proporciona información, por ejemplo, el número cero) más sin embargo el código está adecuado para evitar este tipo de situaciones. Se recomienda al lector comenzar a indagar para encontrar cual es la pieza de código encargada de esta tarea de verificación.

3.4 Ejercicios propuestos

1. Crear un programa que reciba una serie de números y calcule el sumatorio de todos estos.

2. Escribir una función que calcule cuántos números pares hay comprendidos entre dos números límites (sin incluirlos).

3. Crear un programa que calcule si un número es primo o no.

4. Crear un programa que simule el comportamiento de un reloj digital escribiendo en formato de horas, minutos y segundos, iniciando desde las 00:00:00 horas hasta las 23:59:59 horas. Cuando se ejecute el programa, deberá aparecer en pantalla el transcurso desde las cero horas hasta las 23:59:59. Puede utilizar varias alternativas, desde imprimir cada hora en pantalla, como limpiar esta última cada vez que la hora cambie.

5. Generar los primeros diez números perfectos.

Un Número perfecto es aquel que es igual a la suma de sus divisores, excluyéndose el propio número. Por ejemplo, el número 28 presenta 5 divisores menores y distintos de 28, que son: 1, 2, 4, 7 y 14. Al sumarlos da como resultado 28

6. Según la reseña histórica cuenta el inventor del juego de ajedrez que el rey que le solicitó hacerlo le preguntó cómo quería que este le pagase, a lo cual el inventor le contestó que ubicara un grano de trigo en el primer cuadro del

tablero, en el segundo ubicará el doble del primero, en el tercero el doble del segundo y así sucesivamente hasta completar los 64 cuadrados.

Se debe crear un programa que muestre un tablero (sin líneas) en donde se indique el número de granos de trigo debió colocar el rey por cada cuadro.

Ejemplo:

1	2	3	4	5	6
12	11	10	9	8	7
13	14	15	16	17	18
24	23	22	21	20	19
25	26	27	28	29	30
36	35	34	33	32	31
37	38	39	40	41	42

7. Realizar las siguientes figuras para *n* niveles:

<table>
<tr>
<td>

```
    *
   **
  ***
 ****
*****
```

#2

</td>
<td>

```
*
**
***
****
*****
```

#3

</td>
</tr>
<tr>
<td>

```
  ***
**
  ***
*
  ***
```

</td>
<td>

```
*****
*****
*****
*****
*****
```

#5

</td>
</tr>
<tr>
<td>

```
*
*  *
*  *  *
*  *  *  *
*  *  *
*  *
*
```

#6

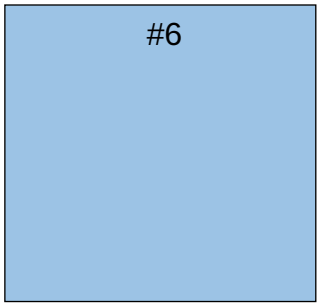
</td>
<td>

```
   *
  * *
 * * *
* * * *
 * * *
  * *
   *
```

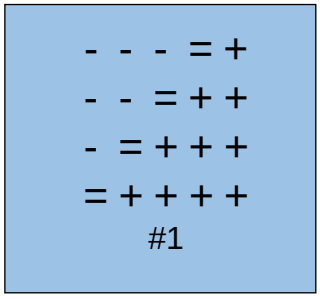
#7

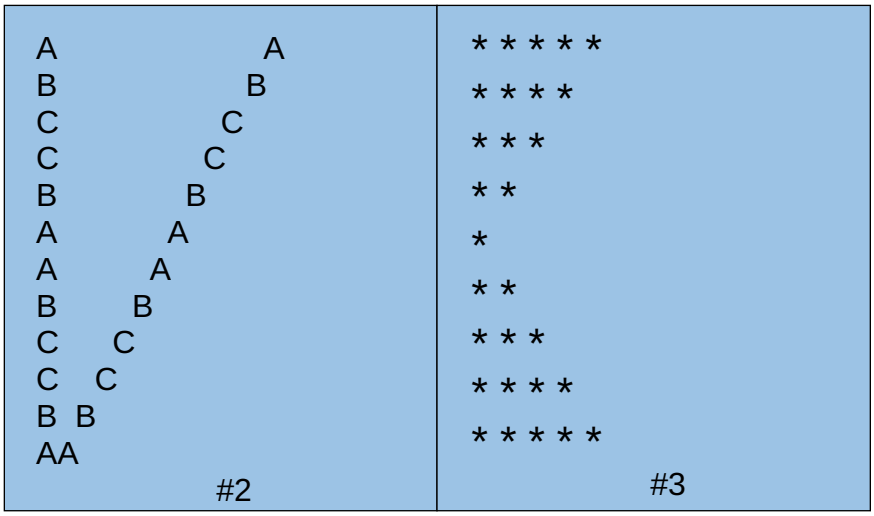
</td>
</tr>
</table>

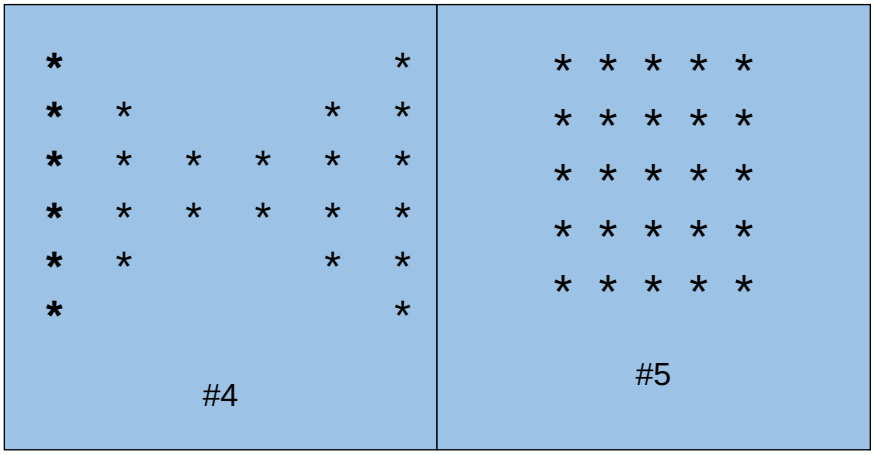
8. Crear el siguiente cuadrado:



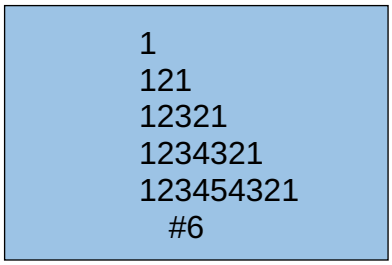
9. Crear el siguiente conjunto de figuras:







10. Leer número e imprimir una pirámide de dígitos:



11. Leer un número positivo e imprimir las sumas de números enteros positivos consecutivos que den el número introducido. Por ejemplo:

    50 = 8 9 + 10 + II + 12
    50 = II + 12 + 13 + 14

4 Funciones

En la programación es muy escuchado el término “Funciones” al momento de escribir código. Hay situaciones en las que se requiere escribir programas que realicen múltiples tareas, cada una mediante una cierta cantidad de pasos de forma ordenada y repetitiva. En este tipo de situaciones se utiliza uno de los lemas más aplicables a todo problema, “*divide y vencerás*”, es decir, descomponer la situación descrita en partes mucho más pequeñas para poder resolver el problema.

Un ejemplo del lema puede ser el siguiente:

“Dados los catetos de un triángulo rectángulo se debe crear un programa capaz de calcular su hipotenusa”

Nota: Siempre, antes de iniciar a resolver un problema se debe identificar lo que se pide, modelar el problema antes de su implementación.

El ejercicio planteado consiste en crear un programa capaz de calcular la hipotenusa de un triángulo rectángulo conociendo los catetos de este. Si se empieza por dividir el problema en sus partes más pequeñas se tendría lo siguiente:

- Recibir y almacenar cada uno de los datos que menciona el ejercicio.
- Encontrar alguna expresión que relacione la información (catetos) con el resultado solicitado.
- Planificar los pasos que se deberán de seguir para encontrar el resultado solicitado.
- Pruebas de escritorio, es decir, probar los pasos generados en el punto anterior antes de pasar a su implementación en código.
- Implementación del punto anterior.
    - Resolver la expresión encontrada para el cálculo del valor solicitado en cada una de las partes que la componen.
- Pruebas.
- Dar solución con la implementación en código del ejercicio.

Cada una de esas pequeñas partes o divisiones mencionadas se conoce como función (o subrutinas). Estas son las encargadas de realizar una acción específica dentro de un conjunto de acciones más amplio. Se componen de órdenes o instrucciones las cuales indican el proceso que se ejecuta cada vez que estas subrutinas son utilizadas. Son una parte fundamental de la programación ya que permiten una codificación modularizada (es decir, por módulos o partes destinadas a una sola tarea), mantener una eficiencia tiempo-espacio, calidad de estética, entre muchos otros componentes.

4.1 Estructura

La creación de funciones en Ruby sigue la siguiente estructura:

```
def name (<lista de parámetros>)
    *Cuerpo de instrucciones
end
```

La expresión *name* indica el nombre de la función, *(<lista de parámetros>)* es el apartado donde se podrán determinar variables externas que se pueden necesitar. Lo anterior es una característica fundamental para las funciones. En algunas situaciones se requieren cálculos con procedimientos distintos a los que ya se encuentran ejecutados por la función en donde se requiere dicho parámetro, por lo que surge la necesidad de tomarlos como parámetros que ingresan en otras funciones y no como expresiones locales de la función.

El **cuerpo de instrucciones* almacena todas las operaciones de las cuales se encargará la función. Es de buenas prácticas relacionar el nombre de la función (*name*) con el procedimiento u objetivo a lograr, por ejemplo, si se desea hacer una función que calcule la media de un conjunto de datos suministrado, lo más ideal es que la función se llame con algún nombre o palabra relacionable a la operación, ya sea usando *media*, *promedio*, etc.

Nota: Para las buenas prácticas en la escritura de código en general se recomienda leer el libro de *Clean Code por Robert C. Martin*

La palabra *end* indica el lugar donde termina la definición de la función. Si se escribe código después de este indicativo, esa parte del código no se ejecutará cuando la función sea ejecutada.

4.2 Funcionamiento

Como ya se indicó en la sección anterior del capítulo, las funciones ayudan a determinar tareas generales a partir de pequeñas partes o divisiones, contribuyendo todas a un objetivo general. Por tal motivo el uso de las funciones tiene ciertas particularidades en la programación; términos y expresiones como “llamado de una función”, “paso de valores”, entre otras, serán algunas de las cuales se aclaran a continuación.

4.2.1 Declaración y uso de funciones

El término “*declaración*” hace referencia al proceso de crear una función, es decir, usar la estructura general de una función definida anteriormente. Para ello se utilizará como ejemplo el siguiente código:

```
1  def sort(set_of_items)
2    length = set_of_items.length - 2
3    for i in 0..length
4      j = i + 1
5      for j in j..(length + 1)
6        if set_of_items[i] > set_of_items[j]
7          auxiliar = set_of_items[i]
8          set_of_items[i] = set_of_items[j]
9          set_of_items[j] = auxiliar
10       end
11     end
12   end
13   print set_of_items
14 end
15
16 sort([6,5,4,3,2,1])
17 sort([456,643,887,362,7378,589957,4,34557684,672,1])
```

Lo anterior es la declaración (o creación) de la función *sort()*. Esta es la encargada de ordenar ascendentemente un conjunto de valores numéricos enteros dados.

Ahora bien, está la definición de la función y fuera de esta se indica su nombre con una característica particular, y es que cada nombramiento contiene un conjunto de datos que precisamente será el evaluado por la función. Toda esta situación se conoce como llamado de una función y el paso de sus argumentos. Es de esta manera en cómo las funciones declaradas se ejecutan.

Como se observa en el ejemplo, si se ejecuta el código el resultado esperado sería el arrojado por la función. Sin embargo, de no ser llamada o nombrada en algún lugar no se retornará ningún valor proveniente de la función.

Con todo ello, se podría re-definir la estructura de las funciones de la siguiente manera:

```
def name (<lista de parámetros>)
        *Cuerpo de instrucciones
end
                                        …

name(<lista de argumentos>) #Aquí se llama, utiliza la función
```

Por otra parte, la nueva estructura define el conjunto de elementos con los que trabajará la función como *<lista de argumentos>*, sin embargo, nótese que en la definición de la función aparece un nombre distinto. Cada uno indica situaciones con características completamente distintas, como se muestra a continuación:

- **Argumento(función):** Son los valores con los cuales se desarrollan las instrucciones almacenadas en la función. Estos valores son asignados en el momento de llamar a una función. Por ejemplo el conjunto de elementos *[6,5,4,3,2,1]* asignado al llamado de *sort()* en la línea 16.

- **Parámetro(función):** Indican el tipo de datos que se pueden usar en el algoritmo que desarrolla la función en su cuerpo de instrucciones. Se construyen al momento de crear la función; almacenan valores cuando esta última es “llamada” y asignada con los argumentos necesarios.

Hay que tener en presente el hecho de que no siempre se tendrán que usar argumentos y parámetros en las funciones, todo depende de si estas necesitan obtener valores externos; en un caso donde no se utilicen, sólo será necesario el llamado.

Nota: A partir de la facilidad proporcionada por la sintaxis de Ruby en los casos en donde no se utilicen parámetros, y por ende argumentos, el llamado a la función podrá hacerse con o sin paréntesis, más sin embargo es recomendable utilizarlos ya que facilitan la comprensión del código.

Ejecutando la función sort()

Al pasarle los argumentos *[6,5,4,3,2,1]* y *[456,643,887,362,7378,589957,4,34557684,672,1]* , el resultado esperado sería:

- [1, 2, 3, 4, 5, 6]
- [1, 4, 362, 456, 643, 672, 887, 7378, 589957, 34557684]

Por ende, la ejecución retorna lo siguiente:

```
$   [1, 2, 3, 4, 5, 6]
    [1, 4, 362, 456, 643, 672, 887, 7378, 589957, 34557684]
```

Detallando las instrucciones ejecutadas por *sort()* se tiene que a nivel general se está ejecutando un algoritmo bastante conocido en el campo de la programación, este es el *bubble sort* u ordenamiento de burbuja. Se basa en comparar uno de los valores proporcionados contra el resto de los elementos en un conjunto de datos determinando cuál de los dos cumple con la condición de ordenamiento (mayor o menor).
Una representación del funcionamiento de *sort()* es la siguiente:

Para un arreglo de tres elementos *[3,2,1]:*

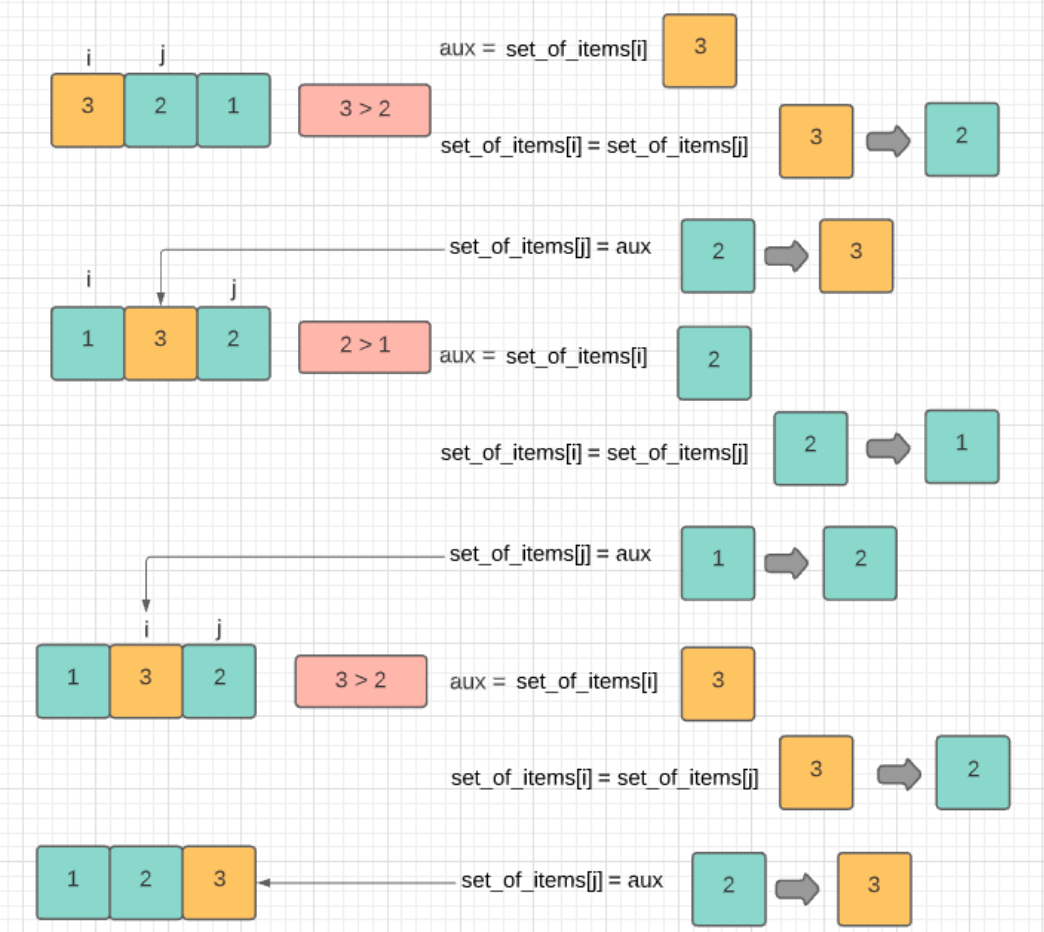


Ilustración 5. Representación del proceso que desarrolla la función sort().

En código se utilizan dos estructuras de iteración *for* que servirán para comparar cada elemento con el resto de los elementos dentro del conjunto. Por otra parte, en el for más interno es donde se realiza la comparación e intercambio entre los dos números analizados. Allí se utiliza el siguiente proceso de intercambio:

- La variable *auxiliar* almacena el valor que cumple la condición analizada, que para el caso del código sería *set_of_items[i]*, en donde *i* guarda la posición en la que se encuentra dicho valor dentro de *set_of_items*
- El ordenamiento es dado en las expresiones *set_of_items[i] = set_of_items[j] y set_of_items[j] = auxiliar* , y básicamente expresan que al acceder al valor almacenado en *set_of_items* dado por las posiciones *i* o *j*, dicho valor debe ser cambiado por el valor que hay en la posición *j* dentro del conjunto de datos o el valor almacenado en auxiliar*,* todo depende según la operación observada.

Nota: Se deja como deber del lector analizar la razón por la cual los límites hasta los cuales deberán de iterar cada estructura *for* son distintos.

4.2.2 Retorno de valores

Las funciones, dentro de su propia definición pueden mostrar información en pantalla. Por otra parte, estas pueden devolver información al lugar en donde fueron llamadas y posteriormente visualizarla.

Para comprender mejor se tiene el siguiente ejemplo:

```
def sort(set_of_items)
        …
end

sort([6,5,4,3,2,1])
```

El código anterior presenta la versión de la función *sort* descrito en la sección anterior y que muestra el resultado desde la función

```
def sort(set_of_items)
        …
        return set_of_items
end

answer = sort([6,5,4,3,2,1])
print( answer)
```

Esta nueva versión retorna el resultado de la función hacía el lugar donde fue llamada para luego almacenar lo anterior en una variable llamada *answer*.

Al ejecutar los dos códigos se podrá esperar que en pantalla se muestre un resultado arbitrario, más el tema al cual se debe poner mayor atención es al hecho que una de las dos versiones utiliza el retorno de información, mientras que otra solo imprime el cálculo por sí misma.

Return

Gracias a la palabra definida por el lenguaje de Ruby, *return*, es posible el *paso de valores* como nueva información para otras variables. Esto permite crear programas más robustos en cuanto a las operaciones que este ejecute.

Para el manejo del *return* se debe de tener claridad en la información que se desea retornar como la forma en que desea guardarla luego de hacer el llamado a la función. Otras características que pueden colaborar en el uso del return son:

- Luego de escribir código justo después de una línea que contenga return, este nuevo código no será ejecutado. Esto es dado gracias a que *return* se indica como una función finalizadora de una función, por ende, se entiende que cuando se llega a ejecutar es porque la función ha terminado con todos sus cálculos.
- *Return* permite devolver datos tanto en cantidad como en naturaleza, es decir, que con ello se pueden retornar uno o más valores de una misma y distinta naturaleza (enteros, flotantes, estructuras, etc.). Este proceso se realiza de la siguiente manera:

```
Def name (…)
                …
        Return información1, información 2, …., información k
end
```

Es en el caso del uso del *return* se hace pertinente que el lector comprenda el tema de *Variables* y su *clasificación* (ver capítulo 2, apartado 2.3), relacionando el tema de expresiones locales y globales.

4.3 Ejercicios propuestos

De modo general, con los capítulos desarrollados hasta este punto se puede proponer que el lector, con el objetivo de afianzar los conceptos indicados aplique las estructuras de funciones a los ejercicios propuestos de cada capítulo anterior.

5 Manejo de tipos de datos compuestos (estructuras de datos)

5.1 Conceptos sobre las estructuras de datos

Una estructura de datos es la herramienta que permite unir, organizar y manipular datos de uno o distintos tipos con el objetivo de proveer un uso más eficiente y comprensible de la información.

Su uso también es atribuido al manejo de grandes volúmenes de información, lo que ayuda en procesos del desarrollo del software como el internet, bases de datos, cálculos de gran magnitud; incluso la implementación de algoritmos, permitiendo estandarizar la solución de un problema en un conjunto de pasos.

5.1.1 Clasificación

La variedad de estructuras se clasifica con base en el tamaño y procesamiento. De lo anterior, se tienen lo siguiente:

5.1.1.1 Tamaño

- **Estático:** Estructura de datos que mantiene un tamaño de almacenamiento fijo en ejecución, es decir, solo permite una cantidad de datos sin posibilidad de nuevos ingresos. Al operar con estructura de este tipo no se podrá cambiar el valor de su tamaño a menos que se utilice una nueva estructura y se re-posicionen los datos almacenados en la estructura inicial.

  Ejemplos: arreglos (vectores, matrices).

- **Dinámico:** Estructura de datos que permite cambiar su tamaño de almacenamiento en ejecución. Al operar con estas estructuras, en medio de la ejecución pueden almacenar o eliminar información configurando a la vez el tamaño que las describe.

  Ejemplos: Listas, pilas, colas, árboles, grafos, deque.

5.1.1.2 Procesamiento

- **Lineal**: Los elementos dentro de estas estructuras se organizan uno tras otro.

  Ejemplos: arreglos (vectores, matrices), listas, pilas, colas, deque.

- **No lineal:** En comparación con el caso anterior, los datos en estructuras no lineales se organizan de forma aleatoria.

  Ejemplos: Árboles, grafos.

Nota: Algunas estructuras de datos teóricamente sólo manejan datos de una sola naturaleza, sin embargo, algunas herramientas permiten implementar estructuras manejando múltiples tipos de datos.

5.2 Arreglos

Los *arrays* (en español como *arreglos*) son estructuras de datos que utilizan la memoria del computador para almacenar información del mismo tipo.

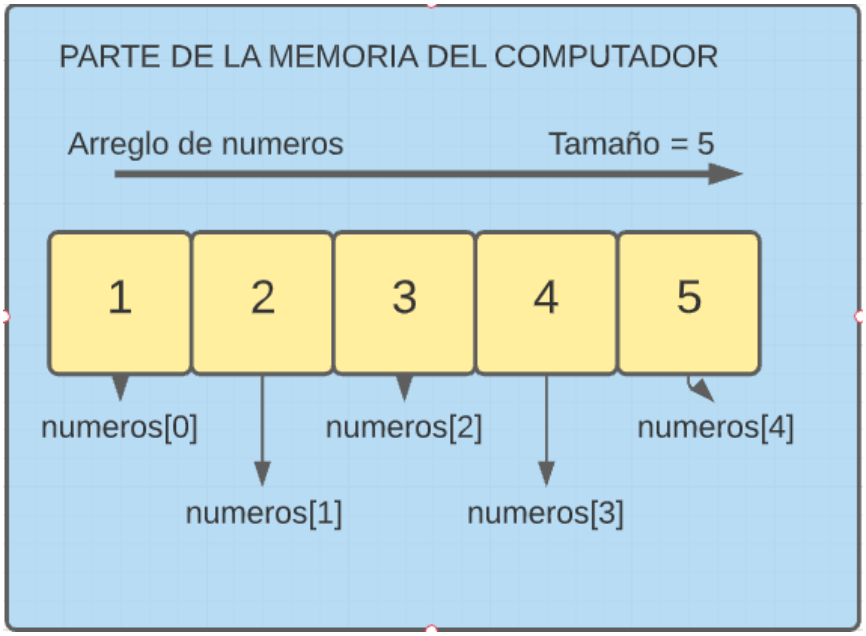


Ilustración 6. Representación gráfica de un arreglo

Se observa en la imagen anterior un conjunto de cuadros almacenados dentro de uno mucho más grande. Son cinco cajas en total más la que contiene a todas. Esta representación indica de forma muy aproximada la abstracción que se hace de un arreglo a la estructura de un computador. El recuadro azul es una pequeña parte de la memoria de un computador mientras que los cuadros contenidos allí son pedazos de dicha parte de memoria tomada que almacenan información que en este caso son números enteros. Observe que cada elemento es contiguo con el siguiente, además de que se describen por un tamaño general (que sería el conteo de todos los cuadros contenidos) más un índice que define su posición con respecto a los otros. Con ellos se puede manipular un arreglo y la información que este almacene.

Son creados en la memoria de un computador de forma consecutiva (un solo bloque de memoria subdividido en más bloques), lo que genera que cada dato se almacene de forma secuencial.
De esta manera se puede manipular un arreglo y la información que este almacene.

Nota: Los arreglos pueden entenderse como variables que almacenan valores de un mimo tipo.

5.2.1 Características de los arreglos

A comparación de otras estructuras, los arreglos disponen de las siguientes particularidades:

- Son estructuras que almacenan información, por ende, necesitan de un tamaño específico.
- Al usar la memoria de un computador, este tipo de estructuras indican la cantidad necesaria a usar para almacenar la información, es por ello por lo que se catalogan como estructuras estáticas.

Nota: Cabe resaltar que, para esta última descripción, el tamaño no es reasignable para un mismo arreglo.

5.2.2 Arreglos en más de una dimensión

Pueden describirse como *arreglos* que contienen *arreglos*, en donde cada arreglo sería una "*dimensión*" según el nombre dado. Una posible representación sería

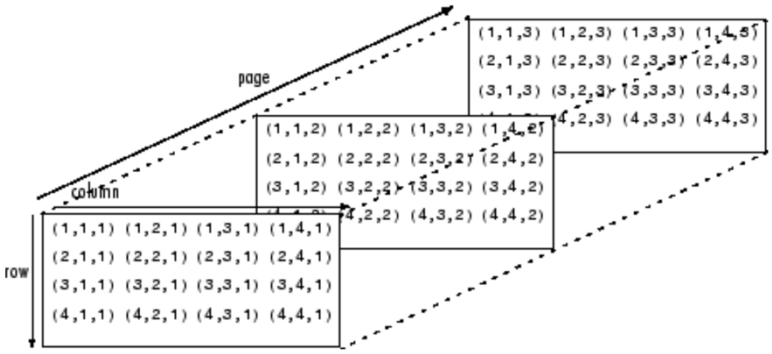


Ilustración 7. Arreglo multidimensional

Fuente: MathWorks

Para mayor comprensión del término "dimensión", se puede pensar en los ejes del plano cartesiano, y que cada eje sería representado por un arreglo, y se sabe que teniendo un plano cartesiano en tres dimensiones (x,y,z) se tendría tres vectores que lo componen. Por otra parte, la ilustración 7 determina el concepto de dimensión con los aspectos de *filas (row), columnas (columns)* y *página (page)* para un vector en tres dimensiones.

Los arreglos multidimensionales pueden poseer un número cualquiera de arreglos representados. Comúnmente se manejan máximo hasta dos dimensiones (o arreglos bidimensionales).

Los arreglos bidimensionales son también conocidos como matrices, es decir, vectores que mantienen una relación similar a la de una tabla de la ilustración 8 donde se observan las filas y columnas.

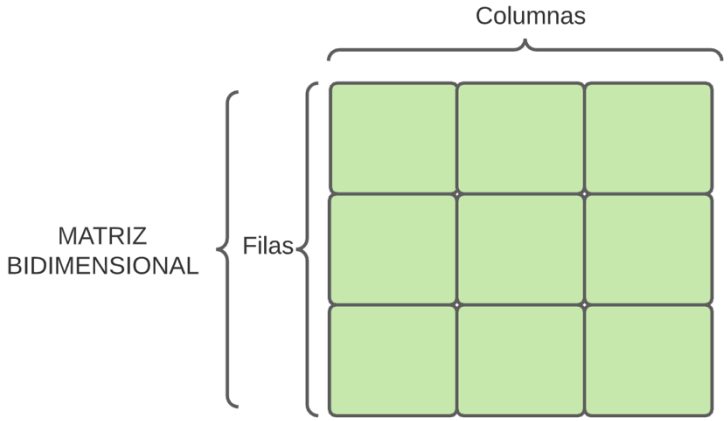


Ilustración 8. Arreglos bidimensionales

Nota: El concepto de matriz también puede expandirse a más de dos dimensiones, como matrices tridimensionales, …, n-dimensionales, entre otras.

5.2.3 Creación e inicialización de arreglos

La expresión *Arreglo* o *Array* es tratada en el lenguaje de Ruby como una clase, concepto que se podrá detallar en los capítulos siguientes. Se menciona lo anterior debido a la forma en como los arreglos son declarados la mayoría del tiempo:

<nombre del arreglo> = Array.new(<tamaño del arreglo>)

Array.new es la clase (representación que define a un arreglo) ejecutando una función que permite crear a nivel de memoria la estructura que se conoce como el arreglo. Un ejemplo podría ser el siguiente:

```
1 a = Array.new(5)
2 print(a)
```

Como resultado se obtiene:

```
> [nil, nil, nil, nil, nil]
```

EL arreglo creado *a* contiene 5 espacios disponibles en memoría, más sin embargo dichos espacios no han sido inicializados, por lo que se muestra por defecto el valor de *nil.*

Por otra parte se puede hacer uso del símbolo ‘[]’:

```
1 a = []
2 print(a)
```

El resultado es:

```
>  []
```

La razón del resultado arrojado es que no hay elementos dentro de *a*, es decir, no hay ninguna expresión dentro de '[]'.

Ahora bien, ya se ha mencionado la inicialización de un vector o, en otras palabras, la inicialización de sus posiciones. Ello es conocido como el proceso de asignarle los valores con los cuales se trabajarán en el programa al arreglo. Para esto se tiene el siguiente ejemplo:

```
1  a = Array.new(5){|i| i.to_s }
2  puts a
```

Lo que se encuentra dentro de las llaves *{…}* es una instrucción de bloque, su concepto no se abarca en estas lecturas más sin embargo puede ser interpretado como un bloque que contiene instrucciones y son pasadas a las función o trozo de código de donde es llamado.

Al ejecutar el código se obtiene lo siguiente:

```
>  0
   1
   2
   3
   4
```

Esos son los valores almacenados en las cinco posiciones definidas en *a.*

Nota: La función *new* ejecutada por *Array* trae consigo otras funcionalidades vistas como argumentos que se le ingresan al momento de ser usada. En algunos de los ejemplos se podrían observar más se recomienda que el lector indague sobre ello.

5.2.4 Arreglos bidimensionales

El lenguaje de Ruby adapta el concepto de una matriz o arreglo bidimensional de " *arreglos que contienen arreglos*" de la siguiente manera:

```
1  a = [[1,2,3],[4,5,6],[7,8,9]]
2
3  for row in 0..a.length-1
4    print("[ ")
5    for col in 0..a.length-1
6      print(a[row][col], " ")
7    end
```

```
8   print("]")
9   puts
10
11  end
```

Como resultado podemos obtener lo siguiente:

```
>  [ 1 2 3 ]
   [ 4 5 6 ]
   [ 7 8 9 ]
```

Observe que directamente se aplicó el concepto dado para arreglos bidimensionales, y para mantener una comprensión más amplía de la estructura y poderla ver como una tabla se decidió por usar variables de *row* y *col*.

5.3 Listas

Las listas al igual que los arreglos son un conjunto de elementos contiguos que bien pueden ser de igual o distintos tipos. Es una estructura muy flexible dentro de la programación ya que siempre permite modificar su tamaño en tiempo de ejecución, es decir, no es necesario indicar un tamaño previo para almacenar datos ya que este se determina a medida que se agregan elementos.

Una lista se puede representar de la siguiente manera:

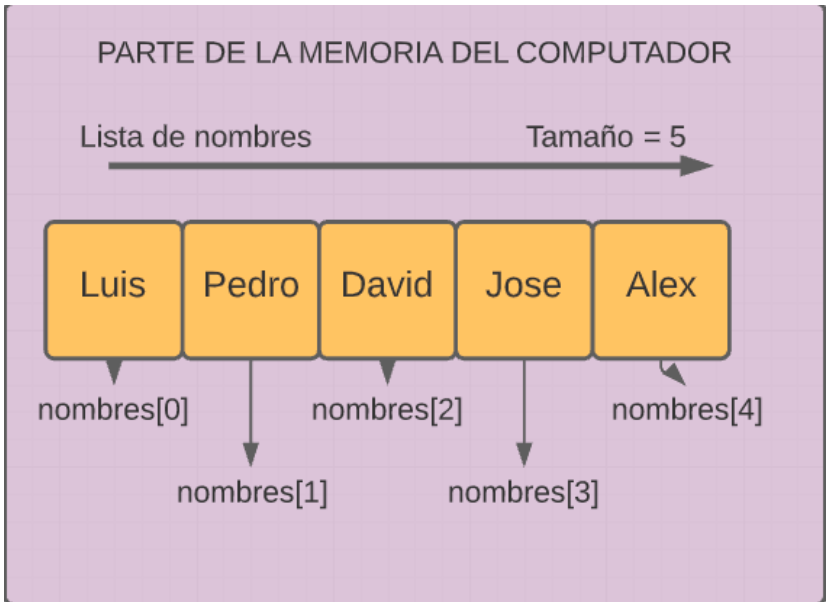


Ilustración 9. Representación gráfica de una lista

5.3.1 Listas desde un enfoque de la memoria de un computador

Gracias a la facilidad de manejo que provee la sintaxis de Ruby, las listas pueden ser comprendidas de una manera particularmente sencilla, sin embargo, en lenguajes enfocados al bajo nivel, es decir, la comunicación directa con el computador usando un lenguaje estricto y funcionalidades más complejas, las listas son estructuras manejadas por la manipulación de un apartado de la memoria de un computador. Este concepto de listas trae consigo nueva terminología como el concepto de punteros, direcciones de memoria, nodos, entre otras las cuales no serán estudiadas en este documento, más si es pertinente que sean mencionadas.

Las listas, desde un enfoque de la memoria de un computador, son bloques que contienen elementos (también llamados nodos) y que se comunican entre sí mediante direcciones las cuales son gestionadas por variables o punteros. Es entonces a partir de este concepto que existe una clasificación de las listas enlazadas según sea la comunicación entre sus elementos, que son la siguientes:

- **Listas doblemente enlazadas:** Estructuras en donde cada bloque de elementos o nodos almacena un puntero con la dirección del elemento anterior y el elemento siguiente, es decir, cada elemento posee dos direcciones de memoria.
- **Lista circular:** Estructuras que comunican su último elemento con el primero.
- **Listas circulares doblemente enlazadas:** Una combinación de los dos tipos de listas anteriores. Consiste en una estructura en que cada elemento contiene la dirección del siguiente y anterior elemento, con la adición de que el último elemento está conectado con el primero.

5.3.2 Operaciones fundamentales sobre listas

De manera teórica, las listas como estructuras manejan ciertas operaciones fundamentales que son:

Operación	Concepto
Size()	Retorna el valor del tamaño de una lista. El tamaño se refiere al número de elementos ocupados allí
Add()	Inserta un elemento (puede ser al final, inicio, o en una posición aleatoria de la estructura)
Remove()	Remueve un elemento (puede ser de cualquier posición)
Get()	Obtiene el elemento de una posición dada
Set()	Cambia el elemento de una posición dada

Ilustración 10. Operaciones fundamentales de las listas

Puede ser el caso que en muchos otros lenguajes de programación se nombren de manera distinta estas operaciones, e incluso usados como generalidad en otras estructuras, pero siempre mantendrán el objetivo de cada una de ellas.

5.4 Ejemplos del uso de Listas y arreglos

Hasta el momento, según la composición de este capítulo las estructuras de listas y arreglos se han tratado diferente. Resulta entonces que el lenguaje de Ruby internamente no difiere en tipos de datos entre listas y arreglos, en su lugar prefiere tratarlos cada uno partiendo de la definición de un arreglo y con la diferencia crucial de que un “arreglo” (encerrado entre comillas para hacer énfasis a la estructura vista en el numeral 5.2) es una estructura que define un tamaño o límite de almacenamiento fijo. Dando continuidad con las “Listas” (siguiendo la misma idea, mencionando a la estructura vista desde el numeral 5.4), son estructuras de tamaño o límite de almacenamiento dinámico, lo que les permite ampliarse cuando se requiere ingresar más información y no hay espacio disponible. Por todo lo anterior, se puede decir con total seguridad de que los ejemplos vistos en la sección 5.3 y 5.4 harían parte del conjunto de listas.

Teniendo en cuenta las condiciones de uso de las listas y arreglos, de ahora en adelante cuando se mencione trabajar con alguna de ellas se sabrá que desde Ruby se utiliza una estructura general para las dos.

Algunos ejemplos propuestos son los siguientes:

1) Generar un programa que permita revertir los elementos de una lista con:
    a. El uso de un vector auxiliar.
    b. Sin el uso de un vector auxiliar.

Desarrollo

Para el caso *a* se tiene el siguiente código

```
1  items = [1,2,3,4,5,6]
2  reversed_items = []
3
4  for item in (items.length-1).downto(0)
5    reversed_items.append(items[item])
6  end
7  print("Lista a invertir -> #{items}\n")
8  print("Lista invertida-> #{reversed_items}")
```

o también:

```
...  …
4    print("Lista a invertir -> #{items}\n")
5    print("Lista invertida -> #{items.reverse}")
```

Ambos dando como resultado:

```
>  Lista a invertir -> [1, 2, 3, 4, 5, 6]
   Lista invertida-> [6, 5, 4, 3, 2, 1]
```

Para el caso *b* se tiene:

```
1   items = [1,2,3,4,5,6,7,8,9,10,11]
2   print("Lista a invertir -> #{items}\n")
3
4   if items.length % 2 == 1
5     middle = (items.length-1)/2
6     increase = items.length
7   else
8     middle = items.length/2
9     increase = items.length
10  end
11  for item in 0..middle-1
12    aux = items[item]
13    items[item] = items[item+increase-1]
14    items[item+increase-1] = aux
15    increase-=2
16  end
17
18  print("Lista invertida-> #{items}\n")
```

Nótese que se puede aplicar para cualquier tamaño de lista, par o impar, por lo que para el caso de la lista presentada en el código se tendría el siguiente resultado:

```
>  Lista a invertir -> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
   Lista invertida-> [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

2) Generar un programa que permita codificar un mensaje de longitud cualquiera mediante los cifrados del Cesar.

Desarrollo

El cifrado de Cesar (nombrado así por su creador Julio Cesar, quien lo usaba para comunicarse con sus generales de manera secreta) podría realizarse de la siguiente manera:

```
1  ALPHABET_SIZE = 26
2
3  def caesar_cipher(string)
4    shiftyArray = [] #almacenará todos los cifrados calculados
5    charLine = string.chars.map(&:ord) #[101,102,...] representación ASCII de string
6
7    shift = 2
8    ALPHABET_SIZE.times do |shift| #Itera |shift|{...} el numero de veces que tiene ALPHABET_SIZE
9      shiftyArray << charLine.map do |c| #operador de asignación <<, se le asigna a shiftArray la rotación de cada letra sumado con la llave o shift
10       ((c + shift) < 123 ? (c + shift) : (c + shift) - 26).chr #ademas de validar que se encuentre dentro del limite de 123 según las repsentaciones ASCII
11     end.join #retorna el nuevo texto cifrado con espacios vacíos entre cada caracter
12   end
13   shiftyArray
14 end
15
16 puts caesar_cipher("testing")
```

Fuente: matugm en github y su tutorial

Al ejecutar el código anterior tenemos lo siguiente:

```
>  testing
   uftujoh
   vguvkpi
   whvwlqj
   xiwxmrk
   yjxynsl
   zkyzotm
   alzapun
   bmabqvo
   cnbcrwp
   docdsxq
   epdetyr
   fqefuzs
   grfgvat
   hsghwbu
   ithixcv
   juijydw
   kvjkzex
   lwklafy
```

	mxlmbgz
	nymncha
	oznodib
	paopejc
	qbpqfkd
	rcqrgle
	sdrshmf

3) Desarrollar un algoritmo que permita crear un cuadrado mágico (donde la suma total de la diagonal principal y secundaria, sus filas y columnas son iguales a 15)

Desarrollo

El código del ejercicio es el siguiente:

```
1  def show_square(square)
2    for row in 0..square.length-1
3      print("[ ")
4      for col in 0..square.length-1
5        print(square[row][col], " ")
6      end
7      print("]")
8      puts
9
10   end
11 end
12
13 def rotate(square)
14
15   rotated_square = [[],[],[]]
16   for row in 0..square.length-1
17     for col in 0..square.length-1
18       if row == col
19         rotated_square[1].append(square[row][col])
```

```
      elsif row < col
            rotated_square[0].append(square[row][col])
      elsif row > col
        rotated_square[2].append(square[row][col])
      end
    end
  end
  return rotated_square
end

def verify(square)
  print("Analizando filas: \n")
  for row in 0..square.length-1
    print("Fila #{square[row]} = #{square[row].reduce :'+'}\n")
  end

  print("Analizando columnas: \n")
  for row in 0..square.length-1
    analyzed_space = []
    for col in 0..square.length-1
      analyzed_space.append(square[col][row])
    end
    # analyzed_space = [square[row][col] for row in 0..square.length-1 ]
    print("columna #{analyzed_space} = #{square[row].reduce :'+'}\n")
```

```
42
43   end
44
45   print("Analizando diagonal primaria: \n")
46
47   analyzed_space = []
48   for row in 0..square.length-1
49     for col in 0..square.length-1
50       if row == col
51         analyzed_space.append(square[row][col])
52       end
53     end
54     # analyzed_space = [square[row][col] for row in 0..square.length-1 ]
55   end
56   print("Diagonal primaria #{analyzed_space} = #{square[row].reduce :'+'}\n")
57
58   print("Analizando diagonal secundaria: \n")
59   analyzed_space = [square[0][square.length-1],square[(square.length-1)/2][(square.length-1)/2],square[square.length-1][0]]
60   print("Diagonal secundaria #{analyzed_space} = #{square[row].reduce :'+'}\n")
61 end
62
63 square = [[1,2,3],[4,5,6],[7,8,9]]
64
```

```
print("Antes de cambiar esquinas \n")
show_square(square)

#Cambiando esquinas asumiendo una matriz cuadrada
aux = square[0][0]
square[0][0] = square[square.length-1][square.length-1]
square[square.length-1][square.length-1] = aux
aux = square[square.length-1][0]
square[square.length-1][0] = square[0][square.length-1]
square[0][square.length-1] = aux

print("Luego de cambiar esquinas \n")
show_square(square)

#Rotando la matriz hacia la izquierda 45 grados aproximadamente
print("Rotando y creando el cuadrado magico con la técnia Lo Shu\n")
magic_square = rotate(square)
show_square(magic_square)

#Verificando resultados
print("Verificando que si es un cuadrado magico: \n")
verify(magic_square)
```

Al ejecutar el código anterior tenemos lo siguiente:

>	Antes de cambiar esquinas
	[1 2 3]
	[4 5 6]
	[7 8 9]
	Luego de cambiar esquinas
	[9 2 7]
	[4 5 6]
	[3 8 1]
	Rotando y creando el cuadrado magico con la técnia Lo Shu
	[2 7 6]
	[9 5 1]
	[4 3 8]
	Verificando que si es un cuadrado magico:
	Analizando filas:
	Fila [2, 7, 6] = 15
	Fila [9, 5, 1] = 15
	Fila [4, 3, 8] = 15
	Analizando columnas:
	columna [2, 9, 4] = 15
	columna [7, 5, 3] = 15
	columna [6, 1, 8] = 15
	Analizando diagonal primaria:
	Diagonal primaria [2, 5, 8] = 15
	Analizando diagonal secundaria:
	Diagonal secundaria [6, 5, 4] = 15

5.5 Pilas y colas

5.5.1 Pilas

Llamada en inglés Stack, son una estructura de datos que almacena y recupera información a través del sistema de acceso LIFO (Last In First Out), donde el último elemento en entrar es el primero en salir.

Las pilas manejan elementos de un solo tipo. Estos son manipulados a través de los extremos de un conjunto total de elementos, es decir, el primer y último elemento. De lo último, se conocerá como TOS (Top of stack o tope de pila) al último elemento, y al primero se le llamará FOS (Front of stack o Frente de la pila).

Lo anterior se puede representar con la siguiente imagen:

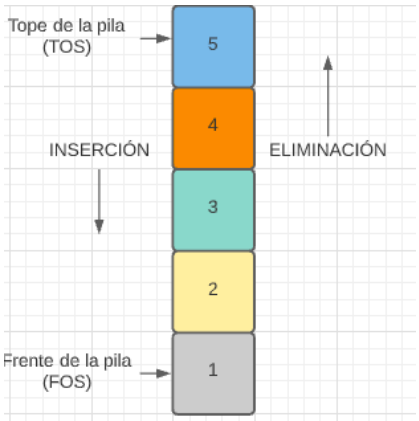


Ilustración 11. Representación física de una pila

Fuente: PROGRAMACIÓN ESTRUCTURADA EN GOLANG, Luis Eduardo Muñoz G.

Estas estructuras pueden ser de utilidad para las siguientes actividades:

- Ayudar en el reconocimiento de lenguajes descritos formalmente
- Situaciones que involucren recursividad.
- Manejo de memoria para ejecución de programas con el Stack de subprocesos.
- Herramientas de organización, distribución y análisis de datos en el desarrollo de aplicaciones.

Nota: Una manera de entender el concepto de pilas es cuando tenemos la necesidad de lavar un conjunto de platos, donde se observa que los últimos platos en la fila tienen la prioridad de ser lavados. Operaciones fundamentales sobre pilas

Se definen las siguientes operaciones fundamentales sobre las pilas o Stack:

Operación	Concepto
Size()	Retorna el valor del tamaño de una pila. El tamaño hace referencia al número de elementos contenidos en esta.
Push()	Inserta un elemento por el extremo final (TOS) .
Pop()	Remueve el Tope de la pila (Tos)
isEmpty()	Retorna un valor booleano que determina si la pila analizada está o no vacía

Tabla 10. Operaciones fundamentales de las pilas

Las operaciones de Push() y Pull() pueden representarse de la siguiente manera:

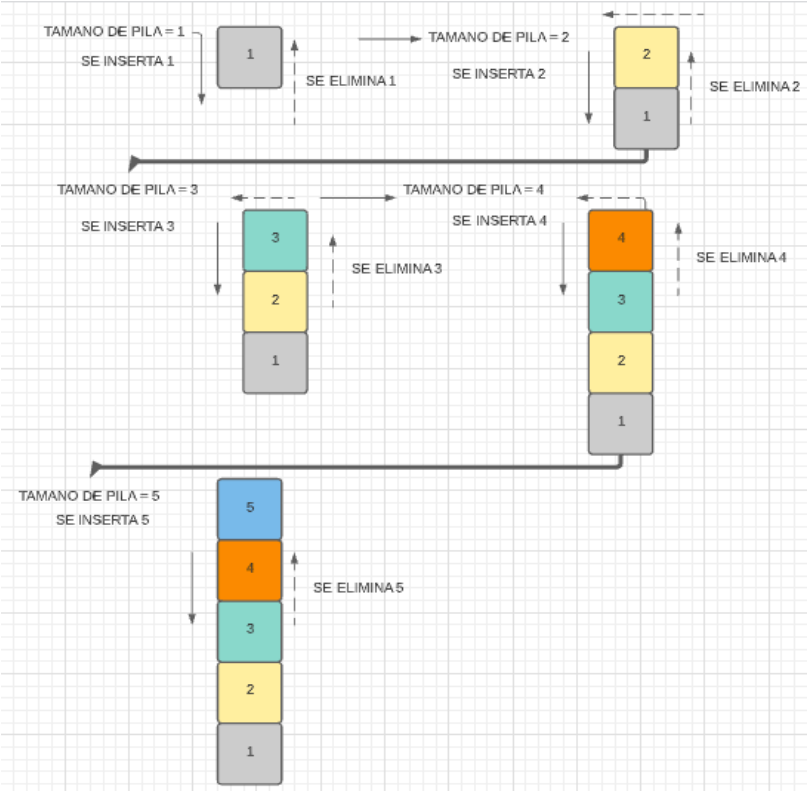


Ilustración 12. Representación de las operaciones Push() y Pul()

Fuente: PROGRAMACIÓN ESTRUCTURADA EN GOLANG, Luis Eduardo Muñoz G.

En la ilustración 11 se presentan las operaciones Push() y Pull(). Las flechas continuas representan Push() (insertar) y las flechas punteadas representan Pull() (eliminar).Nótese que la manipulación de cada elemento perteneciente a la pila se realiza siempre por un mismo extremo.

5.6 Hashes

Nota: Este apartado de Hashes-tablas hash solo será mencionado, y no será temario de este libro.

Los Hashes o tablas hash son estructuras de datos que crean relaciones entre ciertos elementos (denominados llaves) con otros (denominados valores), mediante una función dispersora conocida como función hash y un arreglo de tamaño arbitrario. Puede también definirse a partir del funcionamiento de un diccionario en donde se pueden tener un cierto conjunto de palabras-elementos que traen consigo una definición-asignación.

A manera gráfica, la representación de una tabla hash es la siguiente:

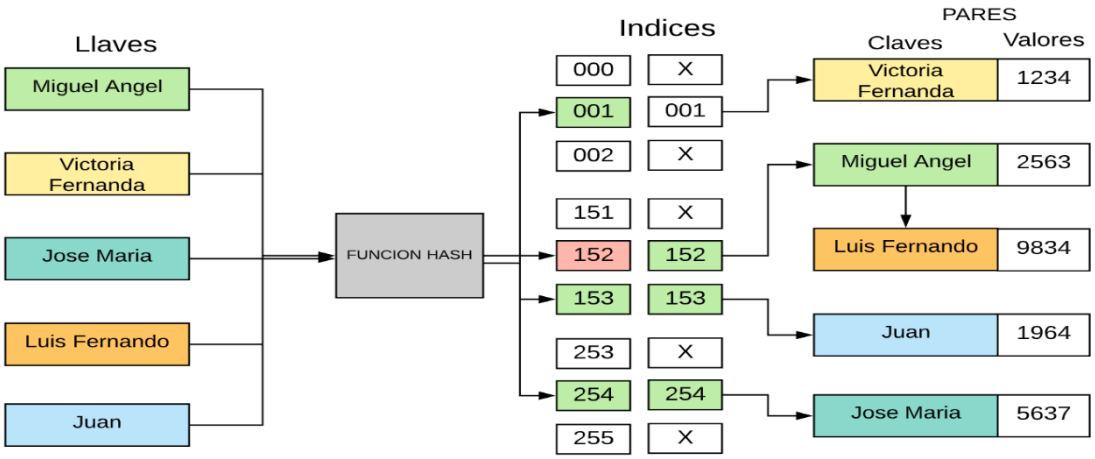


Ilustración 13. Funcionamiento de datos en las tablas hash

Las funciones *Hash* (también conocidas como funciones resumen) utilizan un algoritmo matemático en el que transforman un conjunto de datos en un código alfanumérico, es decir, compuesto de letras, números y otros caracteres, con una longitud fija. Cabe resaltar que dicho código siempre será de una longitud especificada sin importar el tamaño del conjunto de datos analizado.

Gracias a esta función hash se puede destacar la eficiencia de indexación como factor sobresaliente de esta estructura en comparación con otras como las listas (desde su modelo teórico, sin implementación con arreglos).

6 Lista de algoritmos más utilizados en programación

6.1 ¿Qué son los algoritmos?

Adaptando el concepto proporcionado en *Programación de datos con Dr Racket,* en las matemáticas, ciencias de la computación, lógica y relacionadas, un algoritmo se conoce como un *conjunto de instrucciones no-ambiguas, ordenadas y finitas que permiten la solución de un problema o la realización de cierta actividad*. Componen la base que conlleva el proceso de programar, codificar una aplicación, implementación, entre otros.

También son fundamentales en la vida cotidiana para el desarrollo de ciertas habilidades como el razonamiento matemático con las tablas de multiplicar, o incluso el álgebra con las operaciones matriciales. También se incluye en actividades particulares como atarse los zapatos o la manera de realizar un ejercicio.

6.2 Características y clasificación de los algoritmos

Características

Algunas propiedades que caracterizan a un proceso como algoritmo son (4):

- Es preciso y ordenado, avanzando a secuencialmente paso a paso.
- Es definido, independiente del número de veces que se realice, el resultado siempre tiene que ser el mismo.
- Es finito.
- Limita el ingreso de datos entre cada secuencia que realiza.

Clasificación

Los algoritmos se dividen de la siguiente forma:

- Según su sistema de **signos**, es decir, el uso de caracteres verbales, matemáticos y computacionales.
    - **Cualitativos:** Instrucciones que se describen usando palabras. Las recetas o instrucciones serían un ejemplo de ello.
    - **Cuantitativos:** Se usan números y operaciones matemáticas para encontrar un resultado o configurar un procedimiento. Por ejemplo, las operaciones en el cálculo matemático.
    - **Computacionales:** Instrucciones de alta complejidad ejecutadas por una computadora.
    - **No computacionales:** Algoritmos manuales, no necesitan ayudas computacionales, ya sea enfocados al ámbito de optimización,

dinamismo, incremento, búsqueda, entre otros. Suelen ser muy usados por ambientes empresariales y comerciales.

- **Marcaje:** Estudia al cliente, evalúa sus actitudes frente a distintas circunstancias (precios de lanzamiento, rebajas, aumentos, etc.) y crea un sistema que genera las posibles opciones de venta que más se acomoden.
- **Programación dinámica:** Resuelve situaciones en las que se involucran problemas de maximizar o minimizar un valor a partir de un conjunto de restricciones.
- **Vuelta atrás:** Conocido también como *backtracking*, se encarga de analizar toda la ruta de comportamiento de una situación particular y encontrar aquel que cumpla con ciertas condiciones. Por ejemplo, en las operaciones de mercado, los precios, el tráfico de personas en la red, entre otras, se analiza el impacto de estos respecto del tiempo para así determinar si dado su comportamiento en un caso anterior repercute en un aumento o bajada de la cantidad actual.
- **Ordenamiento:** Organizan una secuencia de datos numéricos en un orden específico.
- **Búsqueda:** Encuentra un dato particular en un conjunto de información.

- Según la estrategia usada para el alcance de un resultado son:

  - **Probabilístico:** Estima la probabilidad cierta o incierta de algún proceso o situación particular.
  - **Cotidianos:** Describen las actividades usuales de una persona.
  - **Escalada:** A través de varios resultados, mediante ensayo y error va progresando hasta encontrar un resultado satisfactorio.

6.3 Pseudocódigo

Utilizado para la descripción general de los algoritmos, es un lenguaje de alto nivel, es decir que permite expresar ideas en un lenguaje que sea altamente comprensible para el humano más que para la computadora; este permite usar conceptualmente estructuras vistas en la codificación como las iteraciones, condicionales, funciones, entre otras.

Un ejemplo de ello podría ser la forma en cómo se puede verificar que un elemento dentro de un vector es primo, si cumple se muestra en pantalla, caso contrario su valor cambia a cero y se continúa recorriendo el vector:

```
función  esPrimo(numeroIngresado):
        numeroDeDivisores   =  0
```

por cada numeroEnAumento hasta el rango del intervalo [1,numero +1] hacer
si el numeroIngresado modulo numeroEnAumento es igual a 0 entonces
sume un a numeroDeDivisores

si el numeroDeDivisores es distinto de 2 y el numeroIngresado es distinto de 1 entonces
retorne Falso
de lo contrario
retorne Verdadero

función main():
números = [datos aleatorios en un rango de 0 a 100]

por cada numero en números hacer
si no esPrimo(numero):
índice = índice de numero en números
números[índice] = 0

por cada índice hasta el rango de la longitud de números hacer
sí números[índice] es distinto de cero entonces
mostrar en pantalla números[índice]

Lo anterior puede leerse de manera directa, permitiendo comunicar instrucciones claras de que hacer para lograr el objetivo propuesto, por ende, al intentar pasar loa anterior a código tendríamos lo siguiente:

```
1  def isPrime(number)
2    numberOfDivisors =0
3    for number_eval in 1..number+1
4      if (number % number_eval) == 0
5        numberOfDivisors+=1
6      end
7    end
8      if numberOfDivisors != 2 and number != 1
9        return false
10     else
11       return true
12     end
13
```

```
14  end
15
16  numbers = Array.new(100).map{rand(1..100)}
17
18  for number in numbers
19    if not isPrime(number)
20      index = numbers.index(number)
21    numbers[index] = 0
22    end
23  end
24
25  for index in 0..numbers.length-1
26    if numbers[index] != 0
27      puts numbers[index]
28    end
29  end
```

Como resultado de su ejecución se puede obtener:

```
> 31
  47
  5
  23
  11
  7
  83
  71
  13
  29
```

```
  7
  2
  59
  79
  53
  41
  83
  41
  79
  61
  43
  31
```

	3
	47
	43
	17
	5
	19
	53
	61
	31

	37
	7
	89
	71
	47

6.4 Algoritmos populares

Actualmente aquellos relacionados con el mundo de la programación, en algún momento de su aprendizaje escucharán nombre como *bubble sort, binary search*, *merge sort*, *DFS y BFS*, entre muchos otros. Estos y más son algunos de los algoritmos más utilizados por la industria tecnológica en el desarrollo de software.

6.4.1 Algoritmos de búsqueda (search)

6.4.1.1 Concepto

Partiendo del concepto de algoritmo, esta clasificación se define como instrucciones que buscan un elemento que cuenta con ciertas propiedades dentro de una estructura de datos. Por ejemplo, un caso sencillo sería buscar un número dentro de un arreglo, por otra parte, se pueden tener situaciones mucho más complejas como encontrar la ruta más apta en un conjunto de caminos, o incluso calcular el siguiente mejor movimiento en una partida de ajedrez.

Los algoritmos de búsqueda se dividen según dos particularidades:

- El ambiente proporciona o no información (informado vs no informado). En este caso solo se mencionan algunas de las categorías de no informados
- Si se tienen secuencias de elementos

6.4.1.2 Algoritmos de búsqueda no informados

El término de “no informados” se usa actualmente gracias a su uso en la inteligencia artificial, más sin embargo mantiene su característica de que el ambiente o contexto provea de información como por ejemplo sucede con los árboles binarios (estructuras de datos). Algunos de los algoritmos no informados más utilizados son:

- **Algoritmos heurísticos:** conocidos también como algoritmos aproximados, son un conjunto de instrucciones que calculan una respuesta a un problema dado más no se asegura que la mejor respuesta pueda ser calculada, esto suponiendo que los formas “normales y usuales” no funcionen. Algunos de ellos son:

    - **Greedy algortihm:** Aplicado normalmente a problemas de optimización como conocer el mínimo número de monedas que deben devolverse en un cambio de dinero.

El pseudocódigo del algoritmo es el siguiente:

- Elementos a tener en cuenta:

```
C : conjunto de candidatos
Clen : longitud-tamaño del conjunto de datos en C
S : solución
x : elemento perteneciente a C

S = 0
while (S != solución y Clen != 0) {
        x = C[indice de elemento]
        C = C – {x}
        if (S ∪ {x} es factible) {
                S = S∪{x}

        }
}

if (S es una solución) return S;
else return “No se encontró una solución”;
```

A nivel general, el algoritmo se basa en iterar sobre un conjunto de elementos verificando si alguno de ellos satisface una condición dada (*S ∪ {x}* es factible).

Para entender mejor su funcionamiento, observe la siguiente implementación para la situación dada:

> *Dado un valor y una colección de monedas, el objetivo es encontrar la cantidad mínima de monedas que sumen igual al valor dado. Por ejemplo:*

```
Monedas = {6, 3, 1, 4}

Valor : 23

Cambio : 6  6  6  4  1
```

```
[Note que el cambio [6  6  6  4  1] es el mínimo de monedas de las cuales
su suma da como resultado 23]


Valor  : 13

Cambio:  6  6  1

[Note que el cambio [6 6 1 ] es el mínimo de monedas de las cuales su
suma da como resultado 13]
```

Código

```
1  def minimumNumberOfCoins(coins, numberOfCoins, value)
2
3    if (value <= 0)
4      return
5    end
6
7    coins = coins.sort
8    change = []
9    sum = 0
10   iterations = numberOfCoins - 1
11
11   coin = 0
12
12   while iterations >= 0 && sum < value
13
13     coin = coins[iterations]
14
14     while coin + sum <= value
15
15       change.push(coin)
16
16       sum += coin
17
17     end
18
18
19     iterations -= 1
20
20   end
21
21
     print("\n El valor dado es ", value, ", ")
```

```
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
```

```

  if sum == value
    print("cambio: ", change)
  Else
    print(" No es posible otorgar un cambio total\n")
  end

end

def main()
  coins = [6, 3, 1, 4]
  numberOfCoins = coins.length
  minimumNumberOfCoins(coins, numberOfCoins, 35)
  minimumNumberOfCoins(coins, numberOfCoins, 12)
  minimumNumberOfCoins(coins, numberOfCoins, 8)
  minimumNumberOfCoins(coins, numberOfCoins, 8.5)
end

main
```

Fuente: basado en los aportes de kalicode

Como resultado se obtiene lo siguiente:

```
>  El valor dado es 35, cambio: [6, 6, 6, 6, 6, 4, 1]
   El valor dado es 12, cambio: [6, 6]
```

	El valor dado es 8, cambio: [6, 1, 1]
	El valor dado es 8.5, No es posible otorgar un cambio total

- **A* algorithm:** Plantea su aplicación en los grafos, este permite conocer las rutas de menor coste entre un nodo origen y un nodo destino
- **Escalada simple (hill climbing algorithm):** Ambos algoritmos permiten encontrar una solución óptima definida dentro de un conjunto de elementos, es decir, una solución óptima local. Es un algoritmo iterativo que inicia con una solución arbitraria y comienza a variar incrementalmente un único elemento de dicha solución hasta que no se obtengan más soluciones óptimas.

- **Algoritmos de búsqueda con adversario:** Como su nombre lo indica, son algoritmos dedicados a analizar juegos entre dos personas, donde se podrá analizar cuál será la siguiente mejor jugada. Algunos de ellos son:

  - **Mini-max:** Es un algoritmo recursivo que minimiza la pérdida en juegos con adversario, en este caso se analiza cuál será la siguiente mejor jugada suponiendo que el adversario elije la peor opción para nosotros.

Algoritmos de búsqueda secuenciales

Se usan frecuente en conjuntos de datos como arreglos (vectores), listas, entre otras estructuras relacionadas. Se basa en ir comparando elemento por elemento hasta encontrar el indicado, esto sin importar si el conjunto de datos está o no ordenado. Algunos de ellos son:

- **Búsqueda secuencial ordinaria (lineal):** Consta de recorrer todo el conjunto de datos (de aquí la razón de ser lineal) comparando elemento por elemento con el criterio de búsqueda. La complejidad de este algoritmo suele ser lineal ya que depende del número de elementos existentes en el conjunto de datos, sin embargo esta complejidad puede ser amortizada o incluso cambiada según la posición en donde se encuentre el criterio de búsqueda; pueden existir tres casos, el caso más favorable que es el elemento a buscar situado al inicio del conjunto de datos, un caso medio que es dicho elemento situado en la zona central del conjunto, y el peor caso que es cuando el elemento en búsqueda se encuentra al final del conjunto (estos mismos escenarios pueden ser analizados en otros tipos de algoritmos, incluso mantener la idea de cambio de complejidad).

  Su pseudocódigo es:

  Vector v= [conjunto de datos de tamaño n]

NumeroABuscar = #

por cada numeroEnAumento hasta el rango del intervalo [0,tamaño de v] hacer

si el NumeroABuscar es igual al número en la posición de v[numeroEnAumento] entonces

Retornar un valor indicando que el número a buscar SI está en el conjunto de datos

Si no:

Retornar un valor indicando que el número a buscar NO está en el conjunto de datos

Nota: Se deja a disposición del lector el formular el código respectivo.

- **Búsqueda binaria (binary search):** Este conjunto de reglas permite encontrar un elemento dentro de un grupo de elementos aplicando el principio de divide y vencerás.
  Este algoritmo se aplica a conjuntos de elementos ordenados (de forma ascendente o descendente), reduciendo el número de comparaciones a realizar y por ende generando una complejidad de cómputo agradable en relación tiempo-cantidad de datos.
  Binary search se basa en buscar por mitades el elemento solicitado, es decir, comparar el elemento del medio con el solicitado y si estos dos coinciden se retorna el índice del elemento, caso contrario si el elemento solicitado es menor o mayor al elemento de la mitad se procede a dividir el conjunto de elementos en dos partes y continuar ya sea por izquierda o por derecha según corresponda con la condición anterior respectivamente. En esta nueva fase se repiten los pasos iniciales, continuando de esta manera hasta llegar al momento en donde no se puedan realizar más divisiones.

  Su implementación es la siguiente:

```
1  def binarySearch(elements,item)
2    lower_bound = 0
3    upper_bound = elements.length - 1
4    number_iterations = 0
5
6    while lower_bound <= upper_bound do
7      center = ((upper_bound-lower_bound)/2)+lower_bound
8      if elements[center] == item
9        print("El item buscado es -> #{item} y se encuentra en la posición #{center} \n")
10       print("item: #{item} --- vector[#{center}]: #{elements[center]}")
```

```
      print("\nNumero de iteraciones realizadas -> #{number_iterations}")
      break
    elsif item < elements[center]
      upper_bound = center - 1
    elsif item > elements[center]
      lower_bound = center + 1
    end
    number_iterations += 1
  end
end

vector = []

for position in 0..100
  vector[position] = rand(100)
end

index = rand(100)
item_to_search = vector[index]

print("El vector sin ordenar -> ", vector, "\n")
print("El vector ordenado -> ", vector.sort(), "\n")
print("El elemento a buscar es: ", item_to_search,"\n")
```

```
34 print("----USANDO BINARY SEARCH----\n")
35 print(binarySearch(vector.sort(),item_to_search))
```

El resultado del código anterior es:

```
>  El vector sin ordenar -> [90, 99, 16, 71, 27, 97, 52, 94, 91, 82, 9, 37, 61, 35, 99, 30, 99, 62, 12, 16, 44, 67, 97, 97, 14, 38, 44, 56, 43, 41, 67, 84, 34, 5, 72, 39, 65, 60, 2, 75, 8, 66, 29, 70, 37, 76, 2, 72, 67, 3, 72, 64, 91, 89, 87, 88, 44, 7, 66, 14, 52, 25, 34, 73, 49, 68, 64, 86, 61, 86, 47, 87, 36, 48, 14, 27, 55, 24, 23, 56, 42, 14, 59, 54, 40, 68, 1, 17, 99, 70, 42, 61, 90, 92, 98, 30, 82, 85, 46, 86, 29]
   El vector ordenado -> [1, 2, 2, 3, 5, 7, 8, 9, 12, 14, 14, 14, 14, 16, 16, 17, 23, 24, 25, 27, 27, 29, 29, 30, 30, 34, 34, 35, 36, 37, 37, 38, 39, 40, 41, 42, 42, 43, 44, 44, 44, 46, 47, 48, 49, 52, 52, 54, 55, 56, 56, 59, 60, 61, 61, 61, 62, 64, 64, 65, 66, 66, 67, 67, 67, 68, 68, 70, 70, 71, 72, 72, 72, 73, 75, 76, 82, 82, 84, 85, 86, 86, 86, 87, 87, 88, 89, 90, 90, 91, 91, 92, 94, 97, 97, 97, 98, 99, 99, 99, 99]
   El elemento a buscar es: 30
   ----USANDO BINARY SEARCH----
   El item buscado es -> 30 y se encuentra en la posición 24
   item: 30 --- vector[24]: 30
   Numero de iteraciones realizadas -> 1
```

6.4.2 Algoritmos de ordenamiento

6.4.2.1 Counting sort

Este algoritmo se basa en ordenar un conjunto de elementos, generalmente números según su frecuencia de aparición. Por ejemplo, suponga el siguiente arreglo:

[2,2,3,3,4,1,1,1,1,3,4,5,2,2,2,3]

Se sabe que cada elemento contenido en esta lista es contable, asignándole a cada uno un índice. Por otra parte, hay algunos de esos elementos los cuales están repetidos.

El counting sort propone entonces ordenar este conjunto de elementos a partir de la frecuencia de aparición de cada uno. Este usa una nueva estructura, en este caso otro arreglo en donde se ordenarán las frecuencias del conjunto teniendo como criterio de indexación el valor del elemento de la frecuencia analizada.

A continuación se presenta la implementación en código de este algoritmo:

```
1  def countingSort(elements,k)
2    count = Array.new(k+1).map{0}
3    output = Array.new(elements.length).map{0}
4    for indexi in 0..elements.length-1
5      indexj = elements[indexi]
6      count[indexj] += 1
7    end
8
9    for index in 1..k
10     count[index] += count[index-1]
11   end
12
13   for indexi in (elements.length - 1).downto(0)
14     indexj = elements[indexi]
15     count[indexj] -= 1
16     output[count[indexj]] = elements[indexi]
17   end
18   return output
19
20 End
21
22 print("\nAntes#{[2,9,7,4,1,8,4]} y después", countingSort([2,9,7,4,1,8,4],9), "\n")
23 print("\nAntes #{[3,44,38,5,47,15,36,26,27,2,46,4,19,50,48]} y después ", countingSort([3,44,38,5,47,15,36,26,27,2,46,4,19,50,48],50), "\n")
```

El resultado obtenido es:

```
>
Antes [2, 9, 7, 4, 1, 8, 4] y después [1, 2, 4, 4, 7, 8, 9]

Antes [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48] y después [2, 3, 4, 5, 15, 19, 26, 27, 36, 38, 44, 46, 47, 48, 50]
```

De manera general el algoritmo se compone de tres iteraciones-loops, el primero en donde se encuentra la frecuencia para cada elemento dentro de *elements*. El segundo que permite definir la posición en la cual deberán de ir los elementos en el arreglo de salida, es decir, en este paso se ordena el arreglo. El tercero y último donde se hace el reposicionamiento de elementos.

Es necesario mencionar también que la complejidad de la ejecución de este algoritmo depende del intervalo de números utilizado, es decir, a mayor rango dentro del intervalo de los elementos, mayor podría ser el tiempo que se toma el algoritmo para ordenar el conjunto de elementos.

6.4.2.2 merge sort

Es un algoritmo basado en la filosofía de “divide y vencerás” que se resume en estos dos principios:

1. Divida la lista sin ordenar en *n* sublistas, cada una de las cuales contiene un elemento (una lista de un elemento se considera ordenada).
2. Fusione repetidamente sublistas para producir nuevas sublistas ordenadas hasta que solo quede una sublista. Esta será la lista ordenada.

Para comprender mejor el funcionamiento general, observe la siguiente imagen:

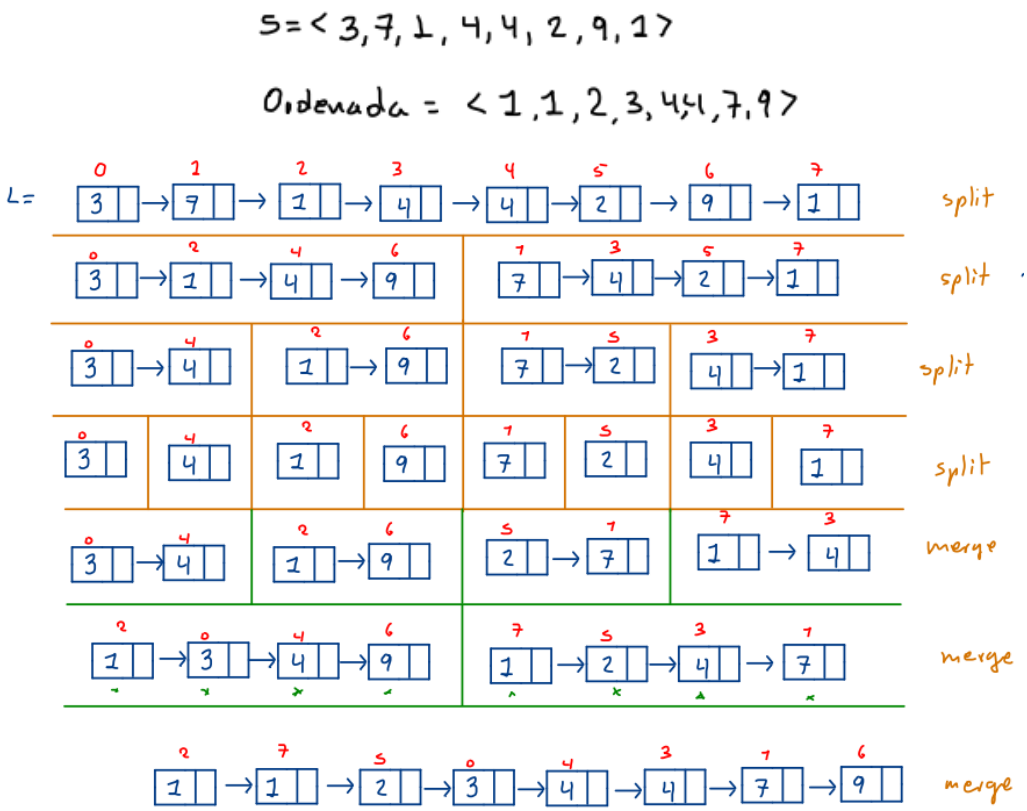


Ilustración 14. Funcionamiento de merge sort
Autor: (Adaptado) Gustavo, 2020. Estructura de datos, Universidad Tecnológica de Pereira

Como se observa en la imagen, partiendo de un conjunto de elementos *s*, se comienza a subdividir este último hasta obtener partes compuestas por un solo elemento (las cuales se consideran que ya están ordenadas). Posteriormente se inicia la comparación entre parejas hasta lograr reagrupar el conjunto, pero esta vez estando ordenado. Se debe resaltar que el paso final en donde se resultan dos mitades ya ordenadas en sí mismas, estas podrán ser comparadas de varias maneras, una de ellas es elemento por elemento.

Actualmente hay varias implementaciones de este algoritmo, una de ellas presentadas aquí es la siguiente:

```
1  def merge_sort(array)
2    if array.length <= 1
3      return array
4    end
5
6    array_size = array.length
7    middle = (array.length / 2).round
8
9    left_side = array[0...middle]
10   right_side = array[middle...array_size]
11
12   sorted_left = merge_sort(left_side)
13   sorted_right = merge_sort(right_side)
14
15   merge(array, sorted_left, sorted_right)
16
17   return array
18 end
19
20 def merge(array, sorted_left, sorted_right)
21   left_size = sorted_left.length
```

```
22  right_size = sorted_right.length
23
24  array_pointer = 0
25  left_pointer = 0
26  right_pointer = 0
27
28  while left_pointer < left_size && right_pointer < right_size
29    if sorted_left[left_pointer] < sorted_right[right_pointer]
30      array[array_pointer] = sorted_left[left_pointer]
31      left_pointer+=1
32    else
33      array[array_pointer] = sorted_right[right_pointer]
34      right_pointer+=1
35    end
36    array_pointer+=1
37  end

38  while left_pointer < left_size
39    array[array_pointer] = sorted_left[left_pointer]
40    left_pointer+=1
41    array_pointer+=1
42  end
43

    while right_pointer < right_size
```

```
44
45      array[array_pointer] = sorted_right[right_pointer]
46      right_pointer+=1
47      array_pointer+=1
48    end
49
50    return array
51  end
52
53  random_array = Array.new(50).map {rand(100)}
54  print("\nBefore #{[3,44,38,5,47,15,36,26,27,2,46,4,19,50,48]} and after ", merge_sort([3,44,38,5,47,15,36,26,27,2,46,4,19,50,48]), "\n")
55  print("\nBefore #{[10,15,1,2,6,12,5,7]} and after ", merge_sort([10,15,1,2,6,12,5,7]), "\n")
56  print("\nBefore #{random_array} and after ", merge_sort(random_array), "\n")
```

Fuente: Megan (mwong068), DEV community Post

```
>  Before [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48] and after [2, 3, 4, 5, 15, 19, 26, 27, 36, 38, 44, 46, 47, 48, 50]

   Before [10, 15, 1, 2, 6, 12, 5, 7] and after [1, 2, 5, 6, 7, 10, 12, 15]

   Before [3, 39, 25, 92, 76, 84, 19, 23, 29, 80, 7, 92, 45, 66, 24, 36, 42, 10, 19, 0, 41, 18, 14, 6, 51, 49, 93, 98, 10, 39, 81, 22, 87, 46, 48, 39, 65, 16, 4, 68, 44, 22, 60, 4, 9, 24, 10, 26, 9, 27] and after [0, 3, 4, 4, 6, 7, 9, 9, 10, 10, 10, 14, 16, 18, 19, 19, 22, 22, 23, 24, 24, 25, 26, 27, 29, 36, 39, 39, 39, 41, 42, 44, 45, 46, 48, 49, 51, 60, 65, 66, 68, 76, 80, 81, 84, 87, 92, 92, 93, 98]
```

6.4.2.3 Quick sort

El quick sort trabaja de cierta manera similar al merge sort, con la diferencia de que toda su acción principal ocurre en la división de elementos y no en la combinación de estos. Los pasos generales que componen a este algoritmo son:

Dado un conjunto de elementos desordenado, se realiza lo siguiente:

1. Escoger un punto de referencia de ordenamiento conocido como **pivote**. Luego se deben recolectar los elementos que sean menores y mayores a este pivote.
   - La elección del pivote determinará cuán rápido/eficaz será el algoritmo. Por costumbre se suele escoger el elemento que se encuentre más a la derecha dentro del conjunto, por ejemplo, una lista como la siguiente [3,2,5,7,2,3,7,4,8,4] tendría como posible pivote el elemento 4 situado en la última posición.
2. Ordenar los elementos menores y mayores al pivote seleccionado anteriormente.
   - Este paso se resuelve de manera recursiva realizando el paso anterior, es decir que para cada sub-lista generada de una división se escogerá un valor pivote que servirá para determinar números mayores y menores a este. Este procedimiento se realiza hasta que las listas resultantes sean de un solo elemento.
3. Combinar todas las sub-listas generadas. Solo combinar ya que es lo único que queda por realizar, esto gracias a que en cada división se determinó que parte era mayor y que parte era menor.

La implementación de este algoritmo con su ejecución es la siguiente:

```
1  def quick_sort(array, first, last)
2    if first < last
3      j = partition(array, first, last)
4      quick_sort(array, first, j-1)
5      quick_sort(array, j+1, last)
6    end
7    return array
8  end
9
10 def partition(array, first, last)
11   pivot = array[last]
12   pIndex = first
13   i = first
14   while i < last
15     if array[i].to_i <= pivot.to_i
16       array[i], array[pIndex] = array[pIndex], array[i]
         pIndex += 1
17     end
18     i += 1
19   end
20   array[pIndex], array[last] = array[last], array[pIndex]
21   return pIndex
```

```
22 end
23
24
25 random_array = Array.new(50).map {rand(100)}
26 print("\nBefore #{[3,44,38,5,47,15,36,26,27,2,46,4,19,50,48]} and after ",
   quick_sort([3,44,38,5,47,15,36,26,27,2,46,4,19,50,48],0,
   [3,44,38,5,47,15,36,26,27,2,46,4,19,50,48].length-1), "\n")
27 print("\nBefore #{[10,15,1,2,6,12,5,7]} and after ",
   quick_sort([10,15,1,2,6,12,5,7],0,[10,15,1,2,6,12,5,7].length - 1), "\n")
28 print("\nBefore #{random_array} and after ",
   quick_sort(random_array,0,random_array.length-1), "\n")
```

Fuente: Megan (mwong068), DEV community Post

```
> Before [3, 44, 38, 5, 47, 15, 36, 26, 27, 2, 46, 4, 19, 50, 48] and after [2, 3, 4, 5,
  15, 19, 26, 27, 36, 38, 44, 46, 47, 48, 50]

  Before [10, 15, 1, 2, 6, 12, 5, 7] and after [1, 2, 5, 6, 7, 10, 12, 15]

  Before [55, 43, 81, 97, 43, 52, 20, 16, 66, 60, 14, 62, 74, 64, 47, 78, 90, 90, 57,
  82, 6, 31, 79, 10, 50, 88, 38, 77, 90, 15, 62, 66, 61, 90, 12, 14, 39, 19, 7, 83, 10,
  85, 56, 14, 31, 40, 60, 27, 9, 20] and after [6, 7, 9, 10, 10, 12, 14, 14, 14, 15, 16,
  19, 20, 20, 27, 31, 31, 38, 39, 40, 43, 43, 47, 50, 52, 55, 56, 57, 60, 60, 61, 62,
  62, 64, 66, 66, 74, 77, 78, 79, 81, 82, 83, 85, 88, 90, 90, 90, 90, 97]
```

7 POO (programación orientada a objetos)

Este paradigma de programación ha permitido a la industria tecnológica actual desarrollar soluciones de software para problemas de gran complejidad, de los cuales generalmente se relacionan con el cómo entender una situación de la vida real en donde este permite recrear lo que el humano fácilmente puede observar y sentir de manera virtual. Algunos casos de aplicación a nivel global podrían ser la inteligencia artificial y su colaboración en la elaboración de modelos de aprendizaje para un campo en específico como los autos que se conducen por sí mismos (vehículos autónomos), también están los programas que permiten manejar los cohetes construidos por empresas como la NASA o Space X, incluso las soluciones que se han propuesto para el sector del internet y consumo como lo son las páginas web y los datos generados (bases de datos) , así mismos se tienen otros que se consideran actualmente sencillos como un sistema operativo y sus funciones, el uso de un lenguaje de programación, entre muchos otros.

7.1 ¿Qué es la programación orientada a objetos (POO)?

La programación orientada a objetos es la forma cómo se puede escribir un código de manera tal que el programa que se esté desarrollando pueda representar lo más fiel posible una situación en específico, es decir, intenta reflejar el funcionamiento de un momento particular de la vida real. Un ejemplo muy común de ello sería un programa que permita manejar la contabilidad de una empresa de venta de productos. Se entiende que allí participan personas como vendedores, operarios, jefes, y contadores, estos últimos se encargan de registrar las acciones realizadas por los vendedores (quienes interactúan con los clientes) tales como registrar una venta. De esta manera y más suceden un conjunto de situaciones las cuales podrían ser planteadas por medio de este modelo de programación.

Comprendiendo el objetivo de la POO (OOP, siglas en inglés para Object Oriented Programming), se debe conocer las particularidades y funcionamiento de este modelo. Como ya se mencionó, el objetivo es representar una situación de la vida real y esto se logra mediante la *abstracción* de la situación a una forma la cual se pueda entender, en este caso la forma es clasificando todo lo observable como aquello que es una **clase** y aquello que es un **objeto**.

7.2 ¿Qué es un objeto?, ¿qué hay de las clases?

Cada término se puede entender de la siguiente manera:

Clase: Hablamos de una clase como si se tuviera un molde con el cual se pueden elaborar múltiples elementos similares entre sí. De una manera más elaborada se entendería la clase como aquella clasificación de elementos que cumplen con ciertas características en común, pero que cada elemento es diferenciable de otro. Por ejemplo, lo que sucede con los moldes de galletas, si bien se pueden crear lo que se conoce como galletas, una mezcla en su mayoría de harina huevos y azúcar, cada una de estas puede tener características distintas como su ancho y alto, las formas, e inclusive el mismo contenido. Se obtienen en general galletas, solo que cada una tiene características distintas.

Otro ejemplo de clase en la vida real podrían ser los animales, si bien existen muchos tipos y clasificaciones, todos ellos comparten características como el sentido de supervivencia, la acción de comer, sus necesidades fisiológicas, el hecho de que todos deben de permanecer en un ambiente específico para su pleno desarrollo, entre muchas otras. Es entonces, al igual que sucede con el molde de galletas expuesto, cada animal es diferenciable de otros por lo que describen sus características.

- **Objeto:** Las clases se entienden como una clasificación de elementos similares pero que cada uno es diferenciable del otro, por lo tanto, un objeto es entonces el elemento clasificado por dicha clase. Continuando con los ejemplos anteriores para el caso en el que se habla de galletas, si se conoce que la clase es galletas y que cada galleta será diferenciable una de otra por una o más características distintas, entonces se puede afirmar que un objeto

de la clase galleta sería la galleta que está hecha de azúcar y harina de trigo, así como que otro objeto independiente que pertenece a la clase de galletas sería aquella galleta hecha de mantequilla y harina de avena.

Para el caso de los animales, un objeto podría ser un león, otro podría ser un elefante, un pájaro, un perro, un gato, entre otros.

Puede verse que el concepto de clase abarca la generalidad de todo lo que se presenta en una situación de la vida real, mientras que un objeto identifica solo una parte de ella.

7.2.1 Atributos y métodos

Hasta el momento se tienen las herramientas que permiten representar una situación de la vida real de una forma general, más no debe pasar por alto los detalles que en gran medida determinan el comportamiento descrito. Estos términos de atributos y métodos son características propias de los objetos y las clases respectivamente los cuales permiten añadir más información, así como representar relaciones.

- **Atributos:** representan las características de los objetos definidos por una clase. Para el caso en el que se tiene como clase los animales, los objetos que saldrían de allí podrían tener como características su descripción física (color, tamaño, extremidades, si tiene pelaje, garras, cola), su hábitat de desarrollo, su alimento, comportamiento, información acerca de su capacidad y manera reproducción, entre muchas otras.
  Para el caso de la clase galletas, cada objeto (es decir, cada galleta proveniente del molde de galletas) tendría como características el sabor, color, olor, textura, forma, rigidez, ingredientes con los que fue elaborado, tiempo sometido al calor, adiciones, entre otras.

- **Métodos:** se sabe que en la vida real todo puede tener una acción, desempeñar alguna actividad, ejercer ciertas funciones. Los animales tienen funciones/acciones de comer, hacer sus necesidades, atacar, caminar, correr, hacer sonidos. Las galletas por su parte pueden verse con funciones de aportar nutrientes, cambiar de forma a medida que son ingeridas, preservarse en el tiempo modificando ciertas características, entre otras. De lo anterior se puede afirmar entonces que los métodos describen las acciones de cada objeto.

7.2.2 Principios de la programación orientada a objetos

La POO establece ciertos principios que ayudan a determinar de una manera más precisa la representación de la vida real por medio de las clases y objetos. Estos principios son los siguientes:

- **Encapsulamiento:** indica que todos los elementos se deben de encontrar a un mismo nivel de abstracción, es decir, de representación. Menciona también que todo debe de ser correctamente “empaquetado” según sea su nivel de procedencia, por ejemplo, en la clase animales se determinó que cada animal proveniente tendría ciertas características, y que estás características serían únicamente de cada animal, se estaría cumpliendo entonces el principio mencionado.
- **Polimorfismo:** como lo indica su nombre, *poli* hace referencia a “múltiples” mientras que *morfismo* se entiende como “formas”, es entonces este principio el que permite representar casos en donde se tengan múltiples comportamientos e inclusive elementos distintos, pero que puedan ser referenciados con un mismo nombre. Esto se puede ver en el caso de la clase animal, en donde un comportamiento distinto en cada objeto sería los sonidos que realiza, una vaca suena diferente a un perro y a un gato.
- **Herencia:** es una forma de relacionar clases formando jerarquías. Permite crear clases a partir de otras preexistentes cediendo sus métodos y generalmente todos sus atributos. La clase animales puede ser el inicio de una jerarquía mayor, en donde partiendo de esta clase inicial se pueden crear otras que representan por ejemplo a los animales marinos, terrestres, aéreos y anfibios. Estos a su vez servirán para clasificar nuevamente el término “animal” en casos particulares. Mediante esta forma de relación se desarrollan de una manera más clara los principios anteriores de **polimorfismo** y **encapsulamiento.**
- **Principio de ocultación:** se basa en aislar un objeto de otros, esto para evitar accesos y cambios no permitidos. Esta visión es más guiada hacia el desarrollo de código, más a nivel práctico podría entenderse como que los objetos son los únicos quienes pueden acceder directamente a sus características. Por ejemplo, un animal ejerce la acción de comer, cambiar su pelaje, mover las extremidades por su propia cuenta. Por otra parte, si bien el acceso directo de otros objetos o de cualquier otro individuo hacia los atributos de otro está negado, se es permitido proveer información acerca de ellos. Este concepto se podrá asimilar de una mejor forma en los ejemplos de código posteriores.

7.2.3 Ejemplos

A continuación, se mostrarán las fases necesarias para crear una clase:

Definir una clase

```
1 class Animal
2 #Atributos, metodos, ...
3 end
```

Al momento de definir las clases siempre se debe de usar la palabra clave *class* seguido del nombre de dicha clase. Este último siempre deberá de ir en mayúscula

ya que es de esta forma en cómo el lenguaje de Ruby entiende que lo que se quiere es crear una clase.

Definición de atributos

Antes de mencionar los atributos es necesario conocer los tipos de variable que se presentan:

- **Variables locales:** Se presentan en la definición de funciones/métodos. Por lo anterior, siempre desempeñan funciones internas a las cuales no se puede acceder (esto encaminado al principio de encapsulamiento y ocultación). Para su representación se utiliza la escritura lower case o también el uso del guión bajo '_' como prefijo del nombre de la variable.

- **Variables de instancia:** También conocidas como atributos, son las variables que dan las características particulares para cada objeto/instancia de una clase, es decir entonces que los valores que cada una toma serán distintos entre objetos. Para su representación se utiliza la escritura lower case y el uso del signo '@' como prefijo del nombre de la variable.

- **Variables de clase:** Como su nombre lo indica, estas variables estarán presentes en la clase, es decir que todos los objetos provenientes de esta última tendrán esta característica con iguales valores. Para su representación se utiliza la escritura lower case y el uso doble del signo '@' como prefijo del nombre de la variable.

- **Variables de globales:** Este tipo de variables pueden ser usadas en múltiples clases al mismo. Visto de otra forma, son definidas por fuera de una clase y a la par dicha clase puede acceder a esta. Para su Representación se utiliza la escritura lower case y el uso del signo '$'como prefijo del nombre de la variable.

Nota: Las variables globales no serán ejemplificadas en los ejemplos posteriores. Se deja a recomendación del lector.

Ahora bien, ejemplificando los conceptos anteriores se tiene lo siguiente:

```
1  $hunters = 10
2
3  class Animal
4
5    def initialize(typeName, color, food, habitat, years)
6      @typeName = typeName
7      @color = color
8      @food = food
9      @habitat = habitat
       @years = years
```

```
10
11    end
12
13  end
```

Como se puede observar en el código, se han descrito atributos de un animal tales como Nombre de su tipo (typeName), color, comida, hábitat y años. Observe que estos han sido escritos dentro de una función llamada *initialize,* esta hará la función de constructor-creador de objetos, es decir que cada vez que se requiera crear un objeto de la clase animal se deberá de usar esta función.

Creación de objetos

Desde el apartado anterior se observó la creación de un método que ayudaría en la creación de objetos, siendo este el método *initialize*. Ahora bien, en conjunto con otra palabra reservada de ruby, *new,* se pueden crear objetos de la siguiente manera:

*Añadir las siguiente líneas al código mostrado en la sección anterior

```
1  def main()
2    myAnimal1 = Animal.new("Cat", "white", "Fish", "My house", 2)
3    …
4  end
5  main()
```

Primero que todo, se decidió usar una función *main* para mantener allí el núcleo de ejecución de todo el código. Segundo, la línea número 2 indica que se está creando un nuevo objeto de la clase Animal con las características de *“Cat", "white", "Fish", "My house" y 2*, es decir que se creará un gato blanco que vive en una casa, come pescado y tiene dos años, y que toda esa información será asignada a la variable *myAnimal1*.

Gracias a *initialize* y al hecho de que dicho método recibe parámetros los cuales inicializan las variables de instancia, es que es posible crear objetos con más información que los describa. Sin embargo, es posible crear objetos que no utilicen el método anterior, con la diferencia de que estos no tendrán información en sus atributos por lo que de una u otra forma tendrán que poder se reasignados en algún otro momento, por ejemplo:

```
1  class Example
2    def SampleClass
```

```
3     puts "Hi, I'm a Example class object and I have no attributes"
4   end
5 end
6
7 def main()
8   object = Example.new
9   object.SampleClass
10 end
```

Observe que la clase *Example* sólo tiene un método con el cual imprime un texto en pantalla, no posee atributos más sin embargo es válido usar la palabra reservada new. Si en algún momento se crearán atributos en otros métodos distintos a *initialize*, estos deben de ser llamados después de la creación del objeto usando la forma anterior.

Getters y setters para la manipulación de atributos

Estos dos tipos de manipuladores permiten, como su nombre en inglés indica, acceder u obtener (*getters)* y cambiar (*setter*) los valores almacenados en los atributos de un objeto. Estos emplean el principio de ocultamiento mencionado anteriormente que nos indicaba la privacidad que se debe de asegurar con respecto a una clase y sus características internas.

Ruby define estas herramientas de la siguiente manera:

class <nombre de la clase>
attr_reader : <nombre del atributo a leer>
attr_writer : <nombre del atributo a cambiar su valor>
attr_accessor : <nombre del atributo a leer y/o cambiar su valor>
end

- *attr_reader:* Define el acceso a los atributos. Indicando los atributos con esta herramienta genera que se pueda acceder al valor almacenado en ellos en cualquier nstante de tiempo.
- *attr_writer:* Define el acceso y cambio de información de los atributos. Indicando los atributos con esta herramienta genera que se pueda acceder y cambiar el valor almacenado en ellos en cualquier instante de tiempo.
- *attr_accessor:* Define las acciones de las dos herramientas anteriores.

Para la clase usada como ejemplo se tendrían los siguiente getters y setters:

```
1 class Animal
2
```

```
3   #getters y setters
4   attr_reader :typeName, :color, :food, :habitat
5   attr_accessor :years
6
7   …
8
9   end
```

En el Código se definió entonces que solo se podrían leer los atributos de *typeName, color, food* y *hábitat*, mientras que *abilities* y *years* serían atributos los cuales podrían cambiar sus valores en cualquier instante de tiempo, por lo que es necesario conocerlos y actualizarlos constantemente.

Usando las características de *myAnimal1* se obtiene lo siguiente:

```
1   def main()
2     myAnimal1 = Animal.new("Cat", "white", "Fish", "My house", 2)
3
4     print("Mi animal es: #{ myAnimal1.typeName}, #{myAnimal1.color} y le
      gusta comer #{myAnimal1.food}.
5           Vive en #{myAnimal1.habitat} y tiene #{myAnimal1.years}.\n")
6
7     myAnimal1.years += 10
8     print("Han pasado 10 años y ahora mi gato tiene #{myAnimal1.years}.\n")
9     myAnimal1.habitat = "Outside Home"
10    print("Mi gato se ha escapado. Ahora su habitat es #{myAnimal1.habitat}")
11  end
```

```
>   Mi animal es: Cat, white y le gusta comer Fish.
          Vive en My house y tiene 2.
    Han pasado 10 años y ahora mi gato tiene 12.
    /Users/miguellopez/RubymineProjects/More of LIBRO
    RUBY/POO/example.rb:26:in `main': undefined method `habitat=' for
    #<Animal:0x00007fd0f89593d8> (NoMethodError)
    Did you mean?  habitat
          from /Users/miguellopez/RubymineProjects/More of LIBRO
    RUBY/POO/example.rb:33:in `<main>'
```

Observe que de ahora en adelante, para poder acceder a las características definidas de un objeto se debe de usar un punto después de la variable que almacena el objeto, luego se escoge la característica a manipular. Observe por ejemplo cómo se accede a la edad de *myAnimal1* con *myAnimal1.years*.

Por otra parte, siempre se debe de verificar el tipo de acceso definido para cada atributo, como en el caso de *myAnimal1.habitat*, el cual solo se definió para que fuese de lectura más sin embargo en un intento de cambiar su valor almacenado se generó un error (texto subrayado en rojo).

Definición de métodos

También conocidos como funcione, estos pueden ser definidos exactamente igual a como se describen estas últimas en el lenguaje de ruby. Por ejemplo, observe los siguientes métodos:

```
1  $hunters = 10
2
3  class Animal
4    #getters y setters
5    attr_reader :typeName, :color, :food, :habitat
6    attr_accessor :years, :walk, :eat, :run
7
8    def initialize(typeName, color, food, habitat, years)
9      @typeName = typeName
10     @color = color
11
11     @food = food
12
12     @habitat = habitat
13
13     @years = years
14
14     @walk = false
15
15     @eat = false
16
16     @run = false
17
17   end
18
18
19   def animalWalk()
20
20     @eat = false
21
21     @walk = true
       @run = false
```

```
22
23      print("El gato está caminando\n")
24    end
25
26    def animalEat()
27      @eat = true
28      @walk = false
29      @run = false
30      eatenFood = 45
31      print("El gato esta comiendo #{eatenFood} gramos de comida\n")
32    end
33
34    def animalRun()
35      @eat = false
36      @run = true
37      @walk = false
38      print("El gato está corriendo\n")
39    end
40
41    def avoidCazadores()
42      print("El animal debe estar seguro, hay #{$hunters} cazadores cerca")
43    end
44  end
```

```
45
46  def main()
47    myAnimal1 = Animal.new("Cat", "white", "Fish", "My house", 2)
48
49    print("Acciones del animal -> Caminar #{myAnimal1.walk}, Correr #{myAnimal1.run}, Comer #{myAnimal1.eat}\n")
50    myAnimal1.animalWalk()
51    print("Acciones del animal -> Caminar #{myAnimal1.walk}\n")
52    myAnimal1.animalWalk()
53    print("Acciones del animal -> Caminar #{myAnimal1.walk}\n")
54    myAnimal1.animalRun()
55    print("Acciones del animal -> Correr #{myAnimal1.run}\n")
56    myAnimal1.animalEat()
57    print("Acciones del animal -> Comer #{myAnimal1.eat}\n")
58
59    myAnimal1.avoidCazadores
60  end
61
62  main()
```

Los nuevos métodos definidos para la clase *Animal* son animal*Walk(),* animal*Eat(),* animal*Run()* y *avoidCazadores()*, a la vez que se anexaron tres nuevos atributos a los cuales se les definió como herramienta de acceso la de tipo *attr_accesor*, lo que permite conocer y actualizar el valor almacenado en ellos.

Al ejecutar el código anterior se obtiene lo siguiente:

```
> Acciones del animal -> Caminar false, Correr false, Comer false
  El gato está caminando
```

```
Acciones del animal -> Caminar true
El gato está caminando
Acciones del animal -> Caminar true
El gato está corriendo
Acciones del animal -> Correr true
El gato está comiendo 45 gramos de comida
Acciones del animal -> Comer true
El animal debe estar seguro, hay 10 cazadores cerca
```

Se puede observar el cambio de los atributos a medida que se utilizan nuevos métodos. Por otra parte, se logra usar la variable global definida como *$hunters* por fuera de la clase, dentro del último método llamado.

7.2.4 Aplicación de los principios de herencia y polimorfismo

7.2.4.1 Herencia

Recordando la teoría mencionada al respecto de este principio, la *herencia se* basa en adquirir el comportamiento de una clase dentro de otra. Continuando con el ejemplo de la clase animal se podría hacer lo siguiente:

```
1  …#Anexando el código anterior de la clase Animal
2  # La clase Dog (perro) hereda todo el comportamiento de Animal
3  class Dog < Animal
4    attr_reader :owner,:name
5    attr_accessor :health
6
7    def initialize(color,food,habitat, years,name,health,owner)
8      super("Dog", color, food,habitat, years)
9      @name = name
10     @health = health
11
12     @owner = owner
13
14   end
15
16   #sobreescribiendo un método para adaptarlo a la clase a animal
17
18   def animalEat()
19
20     @eat = true
21
22     @walk = false
23
24     @run = false
25
```

```
    eatenFood = 1
    print("El perro esta comiendo #{eatenFood} kilogramo de carne\n")
  end
end


def main()
  myAnimal1 = Animal.new("Cat", "white", "Fish", "My house", 2)
  aDog = Dog.new("Brown","Raw Meat", "Domestic",4,"Thomas", "Healthy", "Miguel Angel")

  #imprimiendo para animal
  print("Existe un animal #{myAnimal1.typeName} que come #{myAnimal1.food} y vive en #{myAnimal1.habitat} \n")
  myAnimal1.animalWalk()
  myAnimal1.animalRun()
  myAnimal1.animalEat()
  #imprimiendo para Perro
  print("Existe un perro llamado #{aDog.name} de color #{aDog.color}, come #{aDog.food} y vive en #{aDog.habitat} con su dueño #{aDog.owner} \n")
  aDog.animalWalk()
  aDog.animalRun()
  aDog.animalEat()


end

```

```
41 main()
```

El resultado de ejecutar el código anterior es:

```
> Existe un animal Cat que come Fish y vive en My house
  El animal está caminando
  El animal está corriendo
  El animal esta comiendo 45 gramos de comida
  Existe un perro llamado Thomas de color Brown, come Raw Meat y vive en Domestic con su dueño Miguel Angel
  El animal está caminando
  El animal está corriendo
  El perro esta comiendo 1 kilogramo de carne
```

Analizando el código anterior se observa la adición de nueva sintaxis:

- Definición de herencia con el símbolo '<':

  Clase que hereda < clase que cede

- El uso de una función reservada de Ruby, super(..)
- La posibilidad de reescribir un método definido en una clase padre (aquella que cede su comportamiento) dentro de la clase hija (aquella que hereda)

El código intenta representar la descripción general de un animal y de la cual por medio del uso del principio de herencia se logra definir a un tipo de animal más específico que a su vez abarca un gran conjunto de elementos.

Dentro de la función main se logra acceder a los atributos y métodos desde un objeto perteneciente a la clase Animal como también acceder a los atributos y métodos otorgados desde Animal hacia Dog, comprobando el correcto funcionamiento del proceso de herencia.

Obsérvese entonces que este tipo de herramientas permiten desarrollar programas más complejos en el sentido de que abarquen múltiples tipos de datos definidos por una persona permitiendo a su vez desarrollar tareas complejas de una manera comprensible.

7.2.4.2 Polimorfismo

Este aspecto de la POO dicta la posibilidad de describir una misma acción para diferentes elementos, por ejemplo, todos los animales pueden hacer un sonido, diferenciándose únicamente en lo que producen.

Dado el ejemplo anterior de herencia, se puede comenzar a hablar de polimorfismo desde el momento en que se produce una herencia de animal hacia otras clases, permitiendo la sobre escritura de métodos, como lo es el caso de animalEat que presenta salidas distintas para los objetos de myAnimal1 *y* aDog.

Dicho lo anterior, si se desean crear más clases las cuales heredarán el comportamiento de Animal, como por ejemplo Bird, Lion, Fish, entre otras, será entonces posible la definición de métodos similares para cada uno, pero con salidas distintas.

REFERENCIAS BIBLIOGRAFICAS

Wikipedia. (2022, 07 14). *Wikipedia the free encyclopedia*. Retrieved from Ruby (programming language): https://en.wikipedia.org/wiki/Ruby_(programming_language)

Seifer, J. (2014, 11 26). *teamtreehouse*. Retrieved from blog.teamtreehouse: https://blog.teamtreehouse.com/coolest-ruby-projects-ever

Wikipedia. (2022, 07 17). *Wikipedia the free encyclopedia*. Retrieved from Algoritmo: https://es.wikipedia.org/wiki/Algoritmo

Dave Thomas, C. F. (2004). *Programming Ruby - The Pragmatic Programmer's Guide.* United states: Pragmatic Programmers.

ruby-doc.org. (n.d.). *core*. Retrieved from ruby-doc.org: https://ruby-doc.org/core-3.0.1/doc

Dane, M. (2018, 06 06). *Ruby programming language*.

Flanagan, D., & Matsumoto, Y. (2008). *The Ruby programming language: Everything you need to know.* California: O'Reilly Media Inc.

Olsen, R. (2011). *Eloquent Ruby (Addison-Wesley Professional Ruby Series).* Manhattan: Pearson Education Inc.

J. Jones, P. (2015). *Effective Ruby: 48 Specific Ways to Write Better Ruby.* Estados Unidos: Pearson Education Inc.

Fitzgerald, M. (2015). *Ruby Pocket Reference: Instant help for ruby programmers.* California: O'Reilly Media Inc.

A. Shaw, Z. (2015). *LEARN RUBY THE HARD WAY THIRD EDITION.* Pearson Education Inc.

Shaughnessy, P. (2013). *Ruby under a microscope: An Illustrated Guide to Ruby Internals.* No Starch Press.

Carlson, L., & Richardson, L. (2015). *Ruby Cookbook: Recipes for Object-Oriented Scripting.* O'Reilly Media Inc.

T. Brown, G. (2009). *Ruby Best Practices:Increase Your Productivity - Write Better Code.* O'Reilly Media Inc.

A. Black, D., & Leo III, J. (2019). *The Well-Grounded Rubyist.* Manning.