Un buen repaso de Linux

Física Computacional, UNAM, 2019.

1 El comienzo

Bienvenidos sean ustedes al curso de física computacional. Al final del curso, todas las notas van a convertirse en un libro que les servirá para prepararse para el examen final. La idea del curso no es sólo que conozcan unos cuantos métodos para resolver problemas de ciencia e ingeniería, buscamos también que entiendan que tener la solución asbtracta no es suficiente y un buen conocimiento de complejidad algorítmica y programación son esenciales para que un desarrollo teórico pueda verse reflejado en experimentos computacionales.

Para esto, lo primero que buscaremos es que sus habilidades en Linux sean particularmente buenas, y nos interesa que entiendan cómo funciona la computadora en general.

2 Un poco de historia

A principios del siglo XIX, *Charles Babbage* tuvo una idea que modificaría al mundo. Su aportación fue la *máquina analítica*. Descrita en 1837, la *máquina analítica* contaba con una unidad lógico-aritmética, un control de flujo, memoria, toma de decisiones y ciclos, con lo que era suficiente para ser Turing-completa.

Curiosamente, ninguna de sus máquinas llegó a ver la luz por ciertos conflictos con su jefe y una falta de interés en los posibles financiadores del proyecto, y sus diseños se quedaron para llegar a la realidad hasta 1941 en Berlín, por *Konrad Zuse*.

Sin embargo, no todo lo que *Charles* hizo quedó frustrado. Antes de la *máquina analítica*, él logró construir una versión funcional de la *máquina de diferencias*, otro de sus inventos.

2.1 Máquina de diferencias

La máquina de diferencias automatizaba un algoritmo muy divertido, conocido como algoritmo de diferencias finitas.

Definición 1 *Sea* $\{x_i, y_i\}_{i=0}^n$ *una serie de pares ordenados. Definiremos el polinomio interpolante de Newton como*

$$N(x) \coloneqq \sum_{i=0}^{n} a_i n_i(x)$$

compuesto por los términos de Newton, dejando que $n_0(x) \coloneqq 1$

$$n_j(x) \coloneqq \prod_{i=1}^{j-1} (x - x_j)$$

¿Cómo encontramos los coeficientes? Aquí es donde saldrá el algoritmo. Notemos que cuando $x = x_0$, se anulan todos los términos menos a_0. Con ello, podemos ver que $a_0 = N(x_0) \coloneqq y_0$ (la última definición es porque el polinomio debe ser interpolante). Para encontrar a_1, debemos notar ahora que tomando $x = x_1$ matamos todos los términos, menos $a_0 + a_1(x_1 - x_0) = N(x_1) \coloneqq y_1$, con lo que tenemos que

$$a_1 = \frac{y_1 - y_0}{x_1 - x_0}$$

Para tratar de encontrar el patrón, veamos lo que pasa con $x = x_2$

$$a_0 + a_1(x_2 - x_0) + a_1(x_2 - x_0)(x_2 - x_1) = N(x_2) \coloneqq y_2$$

Que *simplificando* un poquito se convierte en

$$a_2 = \frac{y_2 - y_0}{(x_2 - x_0)(x_2 - x_1)} - \frac{y_1 - y_0}{(x_1 - x_0)(x_2 - x_1)} = \Big(\frac{y_2 - y_0}{x_2 - x_0} - \frac{y_1 - y_0}{x_1 - x_0}\Big)(x_2 - x_1)^{-1}$$

Bueno, no es muy fácil ver el patrón, por eso les voy a explicar lo que está pasando: este polinomio es parecido al de Taylor, sólo que con diferencias finitas en lugar de ritmos de cambio instantáneos. Además, vemos que si hacemos que la serie $\{x_i\}_{i=0}^n$ esté equiespaciada, nos podemos olvidar de hacer productos y podemos construir el polinomio sin multiplicar. Esto es en lo que se basa el éxito de la máquina de *Charles*, que era capaz de almacenar 8 números de 31 dígitos decimales (con esto se podía calcular de manera exacta un polinomio de 7mo orden).

2.2 Las tres generaciones

Podemos dividir las computadoras en 3 generaciones

- 1937-1946: Llevaron el concepto de la *máquina analítica* a la electrónica, pues *John V. Atanasoff* y *Clifford Berry* implementaron la idea con bulbos (unos 18,000 masomenos). La *ENIAC* es otro ejemplo de esta época. Son computadoras sin sistema operativo, que sólo podían realizar una tarea y a las que el concepto de lenguaje de programación les era ajeno.
- 1947-1962: Estas computadoras son un poco más cercanas a lo que tenemos hoy en día. Cuentan con entrada y salida por medio de cintas y tarjetas perforadas, tienen un sistema operativo y al usar transistores son mucho más pequeñas. En 1951 fue *International Business Machine* (IBM) quien sacó al mercado sus computadoras serie 650 y 700.
- 1963-presente: Pues, estas son las computadoras que utilizamos hoy. Aquí nació MS-Dos, Unix, el *ratón*, las *Macintosh*. Son computadoras que pueden realizar diversas tareas a la vez.

3 Unix

Vamos a entrarle de lleno a Linux.

3.1 Preámbulo

Kurzgesagt, Unix es un sistema operativo que tiene sus orígenes en *Bell Labs*. En aquellos viejos tiempos (1950), no era normal pensar en sistemas operativos multiusuario. *Fernando Corbato*, del *MIT*, fue el líder de un proyecto donde apareció el CTSS (*Compatible Time-Sharing System*), pionero de los sistemas multiusuario. Pasaron unos 15 años antes de que el *MIT*, *General Electric* y *Bell Labs* (propiedad de `AT&T`) gestaran un proyecto que buscaba un sistema operativo más ambicioso que el CTSS, batuizado como *Multiplexed Information and Computing System* (o `MULTICS` para los cuates).

Eventualmente los ingenieros (*Ken Thompson* y *Dennis Ritchie*) perdieron el acceso a `MULTICS` debido a las fuertes sumas de dinero que *AT&T* había gastado en el proyecto (tomando en cuenta la adquisición de un *Mainframe GE-645*) que contrastaban con los resultados del proyecto, con pocos resultados y mucho tiempo de retraso. Los ingenieros comenzaron a extrañar algunas cosas que consiguieron en el proyecto de `MULTICS`, como un jueguito que *Thompson* programó, llamado Space Travel. Algo así como un *Kerbal Space Program* sobresimplificado (juego de creadores mexicanos, por cierto).

No es sorpresa entender que *Bell Labs* básicamente le prohibió a sus empleados seguir gastando el tiempo en la investigación de sistemas operativos, pero eso no detuvo a *Thompson*, *Ritchie*, *Canaday*, *Joseph F. Ossanna*, *Stuart Feldmann*, *Douglas McIlroy* y *Robert Morris*. Con la desaparición del proyecto *MULTICS*, desapareció también la necesidad del *Mainframe* adquirido por la compañía. Correr *Space Travel* en el *Mainframe* era, además, muy costoso. *Thompson* encontró entre las antigüedades de la compañía una computadora `PDP-7`, que tampoco estaba asignada a un proyecto en particular en *Bell Labs*. Curiosamente, en su esfuerzo por migrar su *jueguito* a una computadora que le diera menos problemas que el *Mainframe*, *Thompson* se dio cuenta que podía utilizar esta máquina para diseñar un sistema operativo como *MULTICS* siempre soñó ser.

Este sistema operativo fue compilado en el *Mainframe*, y le dio nueva vida a la `PDP-7`. Sin embargo, la computadora no iba a tener la capacidad de lidiar con el sistema operativo que se volvía más y más complejo. Es un buen momento para mencionar que en esta etapa el sistema aún no podía soportar varios usuarios a la vez, por lo que le dieron el nombre (burlonamente) de *UNICS* (*Un-multiplexed Information and Computing System*, aunque otras fuentes citan *Uniplexed*, es casi lo mismo). Este nombre, sin perder la fonética, evolucionó al ritmo que lo hacía el sistema operativo para quedar en la historia como *Unix*.

De una manera un tanto cómica, los ingenieros de alto nivel de *Bell Labs* tenían que encontrar una modo de seguir desarrollando su sistema operativo y renovar los equipos de cómputo a su disposición sin que los jefes supieran que estaban trabajando en un sistema operativo, así que salieron con una creativa excusa: un proyecto de un editor de texto de alto desempeño (léase, *LibreOffice Writer* de antaño), que como pequeñísimo requerimiento tendría la creación de un sistema operativo para soportar el software (así, en letras chiquitas como si fuera inconsecuente). Los directores de la compañía mordieron el anzuelo, y los ingenieros se hicieron de una PDP-11 en Mayo de 1970.

Los altos mandos de la compañía quedaron más que satisfechos con los resultados del nuevo proyecto, en donde vieron también las múltiples virtudes del nuevo sistema operativo que soportaba su flamante editor de texto. Y así siguió el desarrollo de *Unix*.

Es importante destacar que Unix y Linux **no** son lo mismo. Se suele pensar que `Linux` es alguna especie de *ramificación* de `Unix`, pero no es así. Es un *clon* que se hizo desde cero, una copia. El trabajo fue responsabilidad de *Linus Trovalds*, quien no soportó que un sistema como `Unix` sólo pudiese estar disponible de manera comercial y para empresas del calibre de IBM (con su *IBM AIX* que sigue existiendo hoy en día) o HP (con su *HP-UX*).

No todos los héroes usan capa. Cabe destacar que el nombre de *Linux* no fue acuñado por *Linus*. De hecho, él nunca ha estado contento con el nombre pues lo considera un tanto ególatra. Ya que *Linux* contiene herramientas del movimiento GNU, él mismo propuso llamar al sistema *Lignux*, cosa que también fue tirada a la basura por los mismos impulsores de *Linux*. Yo creo que él merece que el sistema lleve su nombre.

Para los que se pregunten. ¿Y por qué no cubriremos MS-Dos? Fácil. El mismo Microsoft ya se está dando por vencido (aunque lo hacen parecer un cambio de cultura laboral dirigido a la apertura y a mejorar el mundo, esa compañía es y será un representante del capitalismo y es guiada por el dinero, no por principios).

3.2 Las partes de Unix

Si bien Unix tiene cientos de miles de líneas de Código, podemos abstraer 3 partes principales

- **Kernel**: Esta es la capa que está en contacto directo con el *Hardware* de la computadora. Se encarga de manejar administrar los recursos del sistema para uso de las aplicaciones. Lleva el control de tiempos para los procesos (pues también dirige al procesador), y también permite el acceso a comunicación externa como redes, otros dispositivos, archivos, etc.
- **Shell**: Esta es la interfaz entre el usuario y el *kernel*. Shell es un `CLI` (*Command Line Interface*) que interpreta los comandos de un usuario y los ejecuta. El shell por defecto que utilizan las distribuciones de *Linux* es *bash*. El autocompletado con el tabulador es una de las facilidades que el *shell* puede ofrecer al usuario para ejecutar las tareas de manera más eficaz.
- **Programas/Utilidades**: Si bien esto no es *necesario*, las utilidades de *Unix* son programitas que realizan tareas de uso común. Como ejemplo, están los comandos que se encuentran en `/bin` o en `/usr/bin`

Sí, yo lo sé, son cosas muy elementales pero es importante tenerlas en la mente para valorar la *simplicidad* de *Linux*.

3.3 Filosofía de *Unix*

La filosofía de *Unix* se compone de los siguientes puntos (éste es el resumen de **Peter H. Salus**, no los 4 puntos originales de **Doug Mcllroy** de *Bell Systems*).

- Haz que cada programa haga una cosa y la haga bien. Busca crear nuevos programas en lugar de crear *funcionalidades* que hagan programas complejos.
- Espera que la salida de cada programa se convierta en la entrada de otro programa. Es decir, procura que la salida sea clara y concisa, pues los programas deben interactuar.
- El lenguaje universal de los programas son las cadenas de texto. Apégate a esto.

Claro, yo añadí algunas cositas pero la esencia de los tres puntos está intacta.

3.4 Estructura de archivos

En *Unix/Linux*, **todo** es un archivo. Repítanlo hasta que lo memoricen. Además de eso, los archivos se encuentran organizados en una estructura de árbol, donde hay **una única raíz** (llamada todo el tiempo `root`, e identificada por la diagonal `/`), y todo nodo contiene al menos dos archivos: se contiene a sí mismo (identificado por un punto `.`) y a su nodo madre (identificado por dos puntos `..`). En el caso de la raíz, no hay nodo madre como tal.

Aquí les va una lista con los principales elementos dependientes de `root` en *Linux*

Nodo	Contenido
`home`	Carpetas principales de usuario
`bin`	Binarios que corren en el sistema
`sbin`	Binarios de superusuario que corren en el sistema
`boot`	Archivos de arranque del sistema
`dev`	De *device*, dispositivos y hardware
`etc`	Archivos varios
`media`	Los dispositivos removibles
`mnt`	Dispositivos montables
`opt`	Archivos de terceros (como `Java`)
`proc`	Archivos falsos de procesos que corren actualmente.
`root`	Diccionario hogar del usuario `root`.
`tmp`	Archivos temporales.
`usr`	Herramientas de usuario (binarios, bibliotecas, íconos, etc.).
`var`	Información variable (como bitácoras, *lock files*, etc.)

La lista no es exhaustiva, pero las carpetas no mencionadas no son de nuestro interés durante el curso.

3.4.1 Rutas relativas y absolutas

Todo archivo tiene una ruta única. Esta ruta se puede representar de dos formas. Supongamos que en mi carpeta personal, `carlos`, hay un archivo llamado `textito.txt`. La ruta absoluta de este archivo sería

```
/home/carlos/textito.txt
```

mientras que la **ruta relativa**, asumiendo que estoy parado en la carpeta `/var/log`, sería

```
../../home/carlos/textito.txt
```

3.5 Comandos básicos

> **Nota:** Para aquellos que estén utilizando un equipo con Linux o con OSx, abrir un terminal va a ser tan sencillo como presionar `ctrl+alt+T` o buscar *terminal* en *spotlight* (`Command+space`). Si están utilizando Windows, no todo está perdido. Bastará con que instalen `WSL` si tienen Windows 10. Con ello, van a poder ejecutar *casi* todos los comandos de `bash`. Como dije con anterioridad, Microsoft está intentando no morir, y como parte de ese esfuerzo, pronto tendrán un Kernel de Linux completo en Windows. Si no saben cómo instalarlo, pueden ver esta guía. Una tercera opción es utilizar `cmder` junto con `git` para tener unos cuantos comandos básicos compilados para Windows.

Vamos a comenzar por recapitular los comandos básicos de edición y navegación por el sistema de archivos

- `man`: El comando más importante de `Linux`, nos muestra la descripción del comando que se le provee como argumento, buscándolo del manual de `Linux`.
- `pwd`: *Path of Working Directory*, nos dice la ruta absoluta en la que nos encontramos.
- `cd`: *Change Directory*, nos permite movernos de directorio, utilizando rutas absolutas o relativas como argumento al comando.
- `ls`: Este comando enlista los contenidos de un directorio. Puede recibir una ruta como argumento. Adicionalmente, aquí están los argumentos más útiles
  - `l`: Despliega el contenido en forma de lista, con información adicional como permisos, tamaño, etc.
  - `a`: Muestra los archivos ocultos también (aquellos que comienzan por punto, lo cual incluye los dos archivos que dijimos todo nodo contiene).
  - `t`: Ordena los archivos en orden de última hora de edición.
  - `1`: Como la lista pero sin detalles (`ls -1` separa con `\n` en lugar de `\s`)
  - `R`: Se ejecuta el comando de manera recursiva en directorios hijo.

- `touch`: Actualiza la estampa de tiempo de los archivos (de acceso más reciente con la bandera `-a` y modificación con la bandera `-m`). Además, sirve para generar archivos nuevos.
- `rm`: Permite eliminar archivos. Por defecto, no elimina directorios, pero se puede forzar con un borrado recursivo (bandera `-r`).
- `mkdir`: Permite crear un directorio.
- `rmdir`: Permite borrar un directorio.
- `cp`: Permite copiar un archivo.
- `mv`: Permite mover un archivo entre dos directorios. El comando se puede utilizar también para renombrar un archivo, como uso indirecto.
- `groups`: Dice los grupos de los que el usuario ejecutor es miembro.
- `whoami`: Imprime el `userid`.
- `passwd`: Les permite cambiar su contraseña.
- `sudo`: Les permite ejecutar comandos como administrador, sólo si están en el **grupo de superusuarios**.
- `chmod`: Cambia los permisos de los archivos. Los permisos son el primer campo que aparece cuando ejecutan `ls -l`. Se ve una cadena de 10 caracteres, algo así como `drw-rwx---`

  $\underbrace{\texttt{d}}_{\text{tipo de archivo}}\quad\underbrace{\texttt{rw-}}_{\text{permiso del creador}}\quad\underbrace{\texttt{rwx}}_{\text{permiso del grupo}}\quad\underbrace{\texttt{---}}_{\text{permiso de cualquiera}}$

  `r` es de lectura,`w` de escritura y `x` de ejecución. normalmente se cambian los permisos con un número, interpretando la tercia como un binario de 3 bits (es decir, los permisos de arriba se leerían como `110111000`, que en grupos de tres es `670`. Léase, los permisos de algún archivo `archivo.txt` se ponen igualitos si ejecutamos `chmod 670 archivo.txt`).
- `bc`: De `basic calculator`, es una calculadora sencilla. El modo *avanzado* se consigue con `-l`.
- `grep`: Es una herramienta de búsqueda de palabras en archivos. Por defecto se utilizan expresiones regulares para la búsqueda, lo que implica que se deben escapar los caracteres `^, {, }, $, +, -` y `.`

Hay algunos cuantos comandos que son de particular utilidad para el manejo de texto

- `cat`: Imprime todo el contenido de un archivo a la terminal
- `head`: Imprime las primeras 10 líneas de un archivo (donde las líneas se terminan por `\n`). La bandera `-n` permite especificar el número de líneas que se quieren extraer.
- `tail`: Imprime las últimas líneas de un archivo, con las mismas reglas que `head`
- `wc`: De *word count*, cuenta el número de líneas, palabras y caracteres. Hay banderas que permiten imprimir sólo uno de estos 3 valores.
- `cut`: Sirve para cortar partes de una línea. Las banderas más usadas son `-d` para especificar un delimitador de campo (entre comillas dobles), y `-f` para especificar qué campos extraer.
- `sed`: Es un editor de *streams* un poco menos complejo que `awk` (el cual en este curso no usaremos). Uno de los usos más comunes, es el siguiente:

  ```
  sed -n 's/patt1/patt2/a' archivo.txt
  ```

  En donde la bandera `-n` dice que sólo las líneas modificadas van a imprimirse, la `s` dice que se va a sustituir el `patt1` por el `patt2`, y la `a` puede ser formada por las siguientes dos opciones:

  - Nada, lo cual reemplaza sólo la primer incidencia
  - Un número para indicar qué número de incidencia se reemplaza
  - La letra `g` para obligar a que todas se reemplazen

Es decir, `sed -n 's/patt1/patt2/2g'` reemplaza desde la segunda incidencia.

Otros comandos divertidos que sirven para manejar el sistema (para esta sección es importante saber que el operador `&` al final de una instrucción manda la ejecución de esta al *fondo*). También es conveniente saber que `ctrl+z` detiene un proceso en la terminal (cuando tardan mucho, eso los pone en pausa a diferencia de `ctrl+c` que mata los procesos).

- `echo`: Repite lo que se le proporciona.
- `cal`: Imprime un mini calendario.
- `date`: Imprime la fecha
- `top`: Este es el administrador de tareas de *Unix*.
- `ps`: Entrega información sobre los procesos que están corriendo.
- `pidof`: Encuentra los números de proceso del comando que se entrega como argumento. Por ejemplo, si *firefox* está ejecutándose, `pidof firefox` entrega su `PID`. Hay un comando similar, `pgrep`.
- `kill`: Permite asesinar procesos, usando el `PID` del proceso. Se obtiene con los comandos anteriores. Este comando permite también mandar mensajes específicos a los procesos, por ejemplo la bandera `-STOP` pone en pausa un proceso en lugar de matarlo, y la bandera `-CONT` reanuda la ejecución del proceso.
- `bg`: Mueve un proceso pausado a seguir ejecutándose en el fondo.
- `fg`: Mueve un proceso pausado a seguir ejecutándose al frente.
- `jobs`: Enlista los procesos que corren en esta sesión.
- `shutdown`: Apaga la computadora cuando se propociona el argumento `now`. La bandera `-r` es muy útil, es para reiniciar la máquina.

Nota: Como una buena nota, los comandos `kill`, `fg` y `bg` aceptan el índice del proceso (como están enumerados al ejecutar `jobs`), siempre y cuando se proporcione acompañado de un `%`. Es decir, `kill %1` mataría al proceso 1 que sale al ejecutar `jobs`.

3.5.1 Conectando comandos

Siguiendo la filosofía de *Unix*, tenemos que aprender a conectar nuestros programitas. Para ello, hay algunos cuantos operadores que debemos aprender, y que *shell* sabrá como interpretar.

- `|` (pipe): Este operador toma la salida del programa que se encuentra a la izquierda y la entrega al programa que se encuentra a la derecha.
- `>`: Este operador dirige la salida de un comando directamente a un archivo.
- `>>`: Este operador dirige la salida de un comando directamente a un archivo, pero no lo sobreescribe, hace una operación de añadir.
- `<`: No suele ser tan usado. Este operador abre el archivo que se encuentre a la derecha y lo pasa como entrada al que está a la izquierda (ojo que normalmente se utiliza `cat archivo.txt`).

Es entonces que podemos comenzar a ver el verdadero poder de *Linux*. Por ejemplo, la calculadora `bc` puede recibir una cadena de operaciones, particularmente que sea resultado de `echo`

```
echo "2+4" | bc
```

Lo cual dará como resultado el número 6.

3.5.2 Wildcards

Los wildcards son caracteres especiales que nos ayudan a *resumir* instrucciones. Veremos más adelante que esto es la versión de juguete de algo mucho más complicado, las expresiones regulares o `regex`. El conjunto común de wildcards es el siguiente:

- `?` El signo de interrogación, que representa un carácter cualquiera.
- `*` Es el signo de interrogación 0 o más veces.
- ∼ La tilde sirve para escapar el signo de interrogación y el asterisco (cuando se quieren buscar como tal en la cadena de caracteres).

Esos son los 3 más usados, pero hay otros 3 que a veces tienen uso:

- `[ ]` Los corchetes son para indicar un rango válido de caracteres o números. Por ejemplo, `m[a-c]l` representa las opciones `mal`, `mbl` y `mcl`.
- {,} Las llaves con valores separados por comas sirven para indicar opciones exclusivas, por ejemplo, `m{a,c}l` representa las opciones `mal` y `mcl`
- `!` El signo de admiración es la negación. Es decir, !a significa cualquier caracter que no sea a. A veces se usa la cuña en lugar del signo de admiración (`^`).

Cuando ustedes colocan un *Wildcard* en una instrucción, es el *shell* el que se encarga de *desenvolver* la instrucción a los casos que sí aplica. Por ejemplo, `rm *.txt` borrará todo archivo que termine con `.txt`.

3.6 Shell Scripting

Todos estos comandos comienzan realmente a adquirir valor cuando los conectamos y los colocamos en un *script* para realizar una tarea compleja y repetitiva.

Los scripts de *shell* permiten hacer flujos con las instrucciones básicas de cualquier lenguaje (`for`, `if` y `while`), sólo que *shell* va a ser muy pedante con la sintaxis.

Un script the *shell* se debe guardar en un archivo (preferentemente con la terminación `.sh`), y se debe ejecutar especificando su ruta relativa comenzando con el directorio actual (`./`). Es decir, si tenemos un script llamado `script.sh`, y estamos parados en el mismo directorio, lo ejecutaremos como `./script.sh`.

3.6.1 Variables

Las variables en *shell* se declaran de manera sencilla: el nombre de la variable, un igual, y luego el valor. **Todo** sin espacios.

```
mivariable=2
```

Las instrucciones sólo terminan con `\n`, no con punto y coma. El valor de una variable se puede obtener con el operador `$`, de preferencia rodeando el nombre de la variable con llaves.

```
mivariable=2
otravariable=${mivariable}
```

Hay algunas variables que el sistema tiene definidas, conocidas como `variables de entorno`. Pueden probar a ejecutar `echo ${PATH}` por ejemplo.

3.6.2 Ejecución de comandos

Si queremos ejecutar un comando, y almacenar su resultado en una variable, podemos utilizar el operador `$()`

```
fechadehoy=$(date)
```

3.6.3 Instrucción `for`

La instrucción `for` se ve masomenos así.

```
#!/bin/bash

for arg in $(seq 1 10)
do
    echo ${arg}
done
```

Me aproveché del comando `seq` para generar una secuencia del 1 al 10. `arg` es un nombre arbitrario, y es una variable que irá tomando los valores de la secuencia en cada iteración.

3.6.4 Instrucción `if`

Las condiciones

```
#!/bin/bash

val=3
if [ ${val} -gt 0 ]
then
    echo "Hurra!"
fi
```

Las condiciones aceptadas dentro de los corchetes son `-lt` (*less than*), `-le` (*less or equal*), `-gt` (*greater than*), `-ge` (*greater or equal*), `-ne` (*not equal*) y `-eq` (`equal`). Para el caso de cadenas de caracteres, las comparaciones se realizan con `==` (iguales) y con `!=` (distinto).

3.6.5 Instrucción `while`

```
#!/bin/bash

count=0
while [ ${count} -lt 10 ]
do
    echo "Hey!"
    count=$(echo "${count}+1" | bc)
done
```

El código anterior escribirá `"Hey!"` 10 veces.

3.6.6 Argumentos

Los argumentos del comando se pueden recuperar con la siguiente sintaxis

- `$@`: Entrega todos los argumentos, sin incluir el nombre del script.
- `$1`: Obtienen el argumento número 1, es decir, el primero. Si se coloca 0 en lugar de 1, se obtiene el nombre del programa, y la regla es *obvia* para los demás números naturales
- `$#`: Se obtiene el número de argumentos, no se toma en cuenta el nombre del script.

3.6.7 Acceso remoto

Un comando que usarán con frecuencia (espero) será `ssh`, el cual permite iniciar un *shell seguro* en un equipo remoto. Basta con poner como argumento el nombre de usuario y la dirección del equipo, unidos por un `@` para iniciar la sesión. Por ejemplo

```
ssh carlos@127.0.0.1
```

intentará iniciar la sesión con el usuario carlos, en el equipo con dirección `IPv4` 127.0.0.1.

Existe también, por ejemplo, el comando `scp` que permite mover archivos entre computadoras (muy útil!). Un ejemplo de uso es el siguiente.

```
scp archivito.txt carlos@127.0.0.1:/home/carlos/mis_docs/
```

que copiará el archivo `archivito.txt` a la carpeta `mis_docs` en el directorio hogar del usuario `carlos`.

Ejercicio 1: Escriba un comando con wildcards que borre todo archivo que contenga la palabra `data` y termine en `.png`

Ejercicio 2: Cree un comando que tome todos los datos del procesador (localizados en el archivo `/proc/cpuinfo`) y cambie todas las letras `o` y `a` por `e`. Ejemplo: `murciélago` se convierte en `murciélege`. Superpuntos bonus si hacen que se conserve la fonética (que en ese caso debería ser `murciélegue`).

Ejercicio 3: Haga un script que reciba una cantidad cualquiera de argumentos, y que imprima los argumentos que tengan exactamente 7 caracteres (usando una línea por argumento que cumpla la condición).

Ejercicio 4: Haga un script que tome un argumento (que puede asumir será una palabra únicamente con minúsculas, sin espacios), e imprima en pantalla el total de la suma del valor de los caracteres, donde definimos el valor de los caracteres como el natural correspondiente a la letra en orden alfabético (es decir a=1, b=2, c=3, etc.). Se pueden utilizar archivos temporales auxiliares.