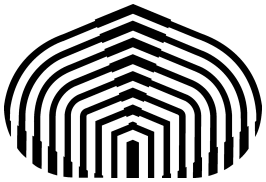

Universidad Simón Bolívar
Departamento de Computación y Tecnología de la información
CI-2692- Laboratorio de algoritmos II
Enero-marzo de 2014

Laboratorio - Semana 2

El objetivo de este laboratorio es dar una introducción al curso y repasar brevemente los conocimientos adquiridos en el curso de algoritmos I, CI-2691.

Indentación:

En Python no se tienen paréntesis, corchetes o palabras claves, como *begin* o *end* para delimitar el comienzo y final del un programa. Tampoco se necesita delimitadores como punto y coma para finalizar las instrucciones. Lo que permite organizar los bloques de código es la indentación.

La indentación se refiere al espacio en blanco o sangría dejado al comienzo de la línea con el propósito de hacer más legible el código. En el caso de Python, la indentación además indica un bloque de instrucciones. Por ello, se debe agregar indentación cada vez que se abre un nuevo bloque o sección de código, es decir, una lista de instrucciones en secuencia dentro de una acción compuesta. Los editores que reconocen la sintaxis del lenguaje de programación suelen ayudar a tener una buena indentación. Si un conjunto de instrucciones está indentado con la misma sangría, esto forma un bloque con el mismo nivel hasta el fin de archivo o hasta que se encuentre una línea con menor indentación.

Hay varios opciones para indentación, pero el estándar es tener cuatro espacios por nivel de indentación. A continuación se coloca un ejemplo de indentación:

```
if condición booleana del condicional:
    # Comienzo del Primer Bloque Interno
    if condición booleana del condicional anidado:
        # Segundo Bloque (dentro del condicional anidado)
        print x
    # de vuelta al bloque anterior
  # Otras instrucciones del primer bloque.
# Continuación del programa
```

Documentación:

Los comentarios deben ser usados para dar una visión del código y proporcionar información adicional que no se encuentra disponible en el mismo. Los comentarios sólo deben contener información necesaria para leer y comprender el programa. Por ejemplo información sobre cómo se debe compilar o ejecutar el programa o en que directorio se encuentra no debe ser incluida como comentario. Se debe incluir información acerca de decisiones de diseño del programa que no sean triviales o que no sean obvias al leer el código.
Debe tenerse el cuidado de no repetir información que ya esté presente en el código y pueda ser obtenida claramente a partir de éste.
Para colocar un comentario en la línea actual utilice el símbolo numeral, '#'.

Ejemplo:

```
# Esto es un comentario desde el comienzo de línea
x = 3     # esto es un comentario después del código.
```

Para comentarios más largos o para una documentación completa, especialmente al comienzo de un archivo .py, utilice la triple comilla doble. Todo lo que quede encerrado entre la triple comilla será ignorado por Python.

Ejemplo:

```
""" Un ejemplo de código en python.
    Note que las triple comillas permiten tener varias líneas y todas
    son ignoradas.
"""
x = 3
```

Tamaños de Línea:

Es recomendable que todas las líneas queden a un máximo de 80 caracteres, o que por lo general éstas tienden a hacer la lectura del programa complicada y al imprimir el código se dificulta mucho su lectura.

La forma preferida de dividir líneas largas es utilizar la característica de Python de continuar las líneas de forma implícita dentro de paréntesis, corchetes y llaves. Si es necesario, puedes añadir un par de paréntesis extra alrededor de una expresión, pero algunas veces queda mejor usar una división invertida. Asegurate de indentar la línea siguiente de forma apropiada. A continuación se coloca un ejemplo que muestra cómo dividir las líneas de diferentes maneras:

```
if size==’large’:
    if width == 0 and height == 0 and \
       color == 'red' and emphasis == 'strong' or \
       highlight > 100:
       print("Has perdido")
    if width == 0 and height == 0 and (color == 'red' or
                                       emphasis == 'None'):
       print("Has ganado")
```

Se sugiere revisar directamente la guía de estilo citada en las referencias [2], en donde se explican convenciones y recomendaciones para tener una mejor calidad de código.

Condicionales:

El capítulo 4 del manual de Python trata los condiciones, los cuales se introducen aquí con un ejemplo:

```
if x < 0:
     x = 0
     print('Negativo convertido en cero')
elif x == 0:
     print('Cero')
elif x == 1:
     print('Uno')
else:
     print('Más de uno')
```

En este condicional, observamos que comienza con la palabra reservada `if` seguida de una expresión booleana que termina con el símbolo de continuación “`:`” (que sustituye al símbolo “`->`” de GCL). El bloque de instrucciones a realizar cuando la expresión es cierta van en la línea siguiente, indentadas a la derecha con al menos un espacio adicional al comienzo del `if`. Sabremos que las acciones se terminan cuando se regrese al nivel de indentación del `if`. Si hay más condiciones o guardias a verificar, se regresa al nivel de indentación del `if` y se inicia la siguiente guardia con la palabra reservada `elif` (que sustituye al símbolo “`|`” de GCL) seguida de una expresión booleana, de los dos puntos, y en la línea siguiente el bloque de instrucciones (indentadas) correspondientes a esta segunda guardia. Se puede agregar tantas guardias como sea necesario. Finalmente, para garantizar que se cubrieron todos los casos posibles (completitud de las guardias) se regresa al nivel de indentación de `if` y se escribe la palabra reservada `else` seguida de dos puntos y en la línea siguiente las instrucciones correspondientes a este caso. Las secciones `elif` y `else` son opcionales. Note que en Python no es necesario un delimitador final como el `fi` de GCL pues basta con regresar la indentación al nivel del `if` para indicar que éste terminó.

Instrucción Nula:

La instrucción nula (skip de GCL) es útil cuando se combina con el condicional. En Python la instrucción nula corresponde a la palabra reservada **`pass.`** Esta instrucción simplemente no hace nada. Puede ser utilizada cuando se requiere colocar al menos una instrucción desde el punto de vista sintáctico, pero el programa no requiere realizar ninguna acción.

Iteraciones

En GCL las iteraciones están precedidas con la palabra reservada **`do`**, seguida de una condición booleana o guardia que finaliza con el símbolo "**`->`**". A continuación, viene una secuencia de instrucciones que se ejecuta repetidamente mientras que la guardia sea verdadera. Al final del bloque se coloca la palabra reservada **`od`**, para indicar el fin de la iteración.

Es importante recordar que GCL verifica en cada iteración que el invariante sea verdadero, y que la cota siempre sea positiva y decreciente. Los invariantes son aserciones intermedias que se escriben en GCL entre llaves ({}) precedida de la palabra reservada **`inv`**. En el caso de las cotas también se escriben entre llaves precedidas de la palabra reservada **`bound`**.

Invariantes

Son expresiones lógicas que generalmente contienen un cuantificador. Dicha expresión debe cumplirse al entrar al ciclo, y luego de cada iteración. En particular, el invariante se satisface luego de salir del ciclo. Un ejemplo de invariante en GCL, que verifica el cálculo de una suma, es el siguiente

```
{inv 0<=k<=N /\ suma =(%sigma i : 0i<k /\ (i mod 2 = 0): i}
```

El ciclo que satisface el invariante, tiene una variable **`k`**, cuyos valores están entre **`0`** y **`N-1`** inclusive. Cuando el ciclo termina **`k`** tiene el valor **`N`** por ello aparece en el rango del invariante de manera que se satisfaga esta condición a la salida del ciclo. La igualdad que aparece en el invariante a continuación del rango de valores de **`k`**, verifica que la variable **`suma`** tenga el valor de la sumatoria de números pares entre **`k`** y **`N`**.

Cotas

La cota es una función que permite garantizar que el ciclo termina. En cada iteración se verifica el valor de la cota, el cual debe ser menor al valor de la cota en el ciclo anterior. Además se

verifica que dicho valor que sea mayor o igual que cero. Una función de cota para el invariante anterior es por ejemplo:

```
{bound N-k}
```

El valor inicial de esta cota es **N**, pues el valor inicial de **k** es **0**. Luego la cota decrece, a medida que **k** va creciendo en cada iteración. Al finalizar el ciclo, **k** tiene valor **N**, por lo que la cota tiene valor **N - N = 0**. Por ello se cumple que la cota es decreciente y el valor final es mayor o igual a cero.

Ejemplo de Iteración en GCL y traducción a Python

A continuación se muestra el ciclo completo de la iteracion en GCL:

```
k,suma:=0,0;
{inv 0<=k<=N suma =(%sigma i : 0<=i<k /\ (i mod 2 = 0): i}
{bound N-k}
do k < N ->
  if (k mod 2 = 0) ->
     suma:=suma + k
  [] (k mod 2 != 0) ->
     skip
  fi;
  k:=k+1
od
```

La traducción a Python de este ciclo sería:

```
k,suma=0,0
cota = N-k+1

# Verificacion de invariante y cota al inicio
assert( 0<=k<=N and suma == sum ( x for x in range(0,k) if (x % 2 ==0) ) )
assert( cota >= 0 )


while ( k < N ):
  if (k % 2 == 0):
     suma=suma+k
  else:
     pass
  k=k+1

  # Verificacion de invariante y cota en cada iteracion
  assert( 0<=k<=N and suma == sum ( x for x in range(0,k) if (x % 2 ==0) ) )
  assert( cota > N-k+1 )
```

```
  cota = N-k+1
  assert( cota >= 0 )
```

Como en Python no es obligatorio el uso del else, es decir, es opcional, el código puede simplificarse de la siguiente forma:

```
k,suma=0,0
cota = N-k+1

# Verificacion de invariante y cota al inicio
assert( 0<=k<=N and suma == sum ( x for x in range(0,k) if (x % 2 ==0) ) )
assert( cota >= 0 )

while ( k < N ):
  if (k % 2 == 0):
     suma=suma+k
  k=k+1

  # Verificacion de invariante y cota en cada iteracion
  assert( 0<=k<=N and suma == sum ( x for x in range(0,k) if (x % 2 ==0) ) )
  assert( cota > N-k+1 )
  cota = N-k+1
  assert( cota >= 0 )
```

Listas en Python.

En Python las listas pueden ser inicializadas como una secuencia de elementos separados por comas y colocados entre corchetes, como se muestra a continuación:

```
>>> a = [0, 14, 100, 1234]
```

Para accesar los elementos del arreglo individuales del arreglo, se hace de forma similar que en GCL a través del índice, es decir, se coloca una expresión entera encerrada entre corchetes a continuación del nombre del arreglo, siendo el 0 el primer elemento. Por ejemplo, para el arreglo anterior, pruebe en el interpretador de Python las siguientes acciones :

```
>>> a[0]
>>> a[5]  // aquí se produce un error pues el arreglo sólo tiene 4 elementos
>>> a[1+1]
```

En Python también se pueden concatenar arreglos con el operador `+`, y también se puede concatenar **n** veces la lista en sí misma con el operador `*`**.** Por ejemplo, pruebe las siguientes acciones en el interpretador de Python:

```
>>> [1,2,3]+[5,6,7,9]
>>> [5,6,7,9]*2
```

```
>>> [1,2,3]+[5,6,7,9]*2
```

Listas Anidadas en Python.

A través de las listas anidadas se implementan los arreglos multidimensionales de GCL mencionados anteriormente, en Python. Por ejemplo, para inicializar un arreglo de enteros de 3 dimensiones de tamaño 3, similar a una matriz 3x3, podemos realizarlo de la siguiente manera.

```
>>> mat = [
...         [1, 2, 3],
...         [4, 5, 6],
...         [7, 8, 9],
...        ]
```

Para accesar los elementos de un arreglo multidimensional, se deben colocar tantos índices como dimensiones tenga el arreglo, de forma similar que GCL, pero rodeando cada dimensión con los corchetes, siendo el `0` el primer elemento en cada dimensión y `n-1` el último elemento si la dimensión es de tamaño `n`. Por ejemplo, para el arreglo `mat`, pruebe en el interpretador de Python las siguientes acciones :

```
>>> mat[0][0]
>>> mat[1][2]
>>> mat[2][2]
>>> mat[3][3] //en este caso da error, por qué?
```

Ciclos FOR

En GCL solo se utiliza el lazo `do` (mencionado en el pre-laboratorio anterior). Sin embargo, en Python tenemos otra opción de lazo, que es el `for`. Básicamente, el `for` se usa para el recorrido en listas [4]. La sintaxis de este lazo comienza con la palabra reservada `for` seguida de la variable que recibirá el valor de cada elemento de la lista, luego la palabra reservada `in`, y la expresión que pueda ser evaluada como una lista seguida de dos puntos (`:`). Igual que en el lazo `while`, las acciones que se deseen ejecutar en cada iteración, se deben colocar con un nivel de indentación (o sangría) dentro del `for`. Por ejemplo, pruebe el siguiente lazo en el interpretador de Python::

```
>>>for x in range(1,4):
...   print(x)
...
```

De esta forma se tiene dos maneras de implementar el recorrido en arreglos (listas) de Python, dependiendo del ciclo que se utilice: `while` o `for`. El `while` es completamente equivalente al do de GCL. El `for` da un recorrido alternativo sin manejo directo del contador del ciclo. Por ejemplo, para imprimir los elementos del arreglo `a` podemos hacerlo de las siguientes dos maneras:

```
k=0
while (
k  <  4
):

print(a
[k])
  k=k+1
```

```
for  k  in
range(0,4
):

print(a[k
])
```

También se puede utilizar el lazo `for` para la declaración de listas de una manera más abreviada. Su estructura es bastante parecida a los cuantificadores realizados en los laboratorios anteriores.

Por ejemplo, si queremos declarar una lista cuyos elementos sean los números desde el 5 al 10, se escribe:

```
>>> [ x for x in range(5,11) ]
```

Si necesitamos agregarle algún tipo de condición, podemos colocarla al final. Por ejemplo, el cuadrado de todos los números pares desde el 5 al 10. Tenemos:

```
[ x*x for x in range(5,11) if ( x % 2 == 0) ]
```

Ciclos Anidados

Cuando una de las instrucciones que aparece dentro de un ciclo es también un ciclo, estamos antes la presencia de ciclos anidados.

Un caso muy común donde aparecen es en los recorridos de matrices. Por ejemplo el siguiente trozo de programa en GCL muestra la suma de los elementos de una matriz m de tamaño NxN o arreglo bidimensional.

```
f,suma := 0,0;
do f < N ->
```

```
    c:=0;
    do c < N ->
       suma := suma+m[f,c];
       c:=c+1
    od;
    f:=f+1
od
```

En este ejemplo el ciclo externo realiza N iteraciones, y el interno N iteraciones por cada iteración del ciclo externo, por tal razón el número total de iteraciones realizadas en N*N.

Si se usa un while, un trozo de programa en Python equivalente es

```
f,suma = 0,0
while f < N :
  c=0
  while c < N :
     suma = suma+m[f][c]
     c:=c+1
  f=f+1
```

Con el ciclo `for` también se pueden realizar ciclos anidados. El ciclo anterior usando for quedaría

```
suma = 0
for f in range(0,N) :
  for c in range(0, N) :
     suma = suma+m[f][c]
```

Por ello, usando `for` se pueden hacer recorridos sobre listas multidimensionales. Por ejemplo, pruebe en el interpretador de Python las siguientes acciones.

```
>>> mat = [ [1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> for f in range(0,3):
...   for c in range(0,3):
...     print (mat[f][c])
...
```

Invariantes y Cotas en Ciclos FOR.

De igual manera que los lazos `while`, pueden colocarse invariantes y cotas a los lazos `for`. Sin embargo, el uso de la función `range` dentro de un for garantiza que el lazo tiene una función de cota que decrece y es positivo por lo cual no es necesario estas condiciones. Sólo indicar cual es la función de cota y verificar el invariante. Para ello seguiremos el mismo esquema que para el lazo `while`, como se indica a continuación:

```
# Cota N - i
# Invariante
assert ( ... condiciones que se deben cumplir ... )

for i in range (0,N):

  # operaciones del lazo

  # Invariante
  assert ( ... condiciones que se deben cumplir ... )
```

Tipos de Datos Estructurados

En Python, para construir estructuras de datos que agrupen atributos de diferentes tipos se utilizan las clases. Con ellas se puede definir un nuevo tipo de datos, que contiene uno o varios atributos con tipos de datos diferentes.

La clase se define utilizando la palabra reservada `class`, seguida del nombre de la clase, y luego dos puntos (`:`). Luego se colocan los atributos o campos de la clase con un valor inicial que indica el tipo de dicho atributo. Para indicar que los atributos pertenecen a la clase en necesario colocar un nivel de indentación o sangría con respecto a la palabra `class`. Por ejemplo:

```
>>> class Persona:
...     edad = 0       # tipo entero
...     altura = 0.0   # tipo real
...
```

Luego para indicar que una variable es del tipo Persona, se le asigna se utiliza la sintaxis `<var>=<tipo>()`. A continuación se pueden usar o modificar los atributos, accesándolos con la notación punto (`.`), de manera similar como se usaba la parte real e imaginaria de un complejo. Por ejemplo:

```
>>> yo = Persona()
>>> yo.edad = 29      # cambia el valor de la edad
>>> yo.altura = 1.75 # cambia el valor de la altura
```

Para comparar dos estructuras es necesario preguntar individualmente por cada campo, pues cada estructura es un objeto diferente. Por ejemplo realice las siguientes acciones en su interpretador de Python.

```
>>> tu = Persona()
>>> tu.edad = 29
>>> tu.altura = 1.75
>>> yo == tu  # aquí el resultado es false pues son objetos distintos
>>> yo.edad == tu.edad 29 and yo.altura == tu.altura #esto si es true
```

ADVERTENCIA: no es correcto asignar directamente por el nombre una estructura a otra, pues esto genera un efecto de borde que puede producir que su programa no funcione bien más adelante. Observe lo que pasa con las siguientes acciones

```
>>> yo.edad = 15
>>> yo.altura = 1.60
>>> tu.edad = 29
>>> tu.altura = 1.75
>>> yo = tu # Prohibido realizar
>>> yo.altura
>>> yo.edad
>>> tu.altura = 1.45
>>> yo.altura
```

Por ello, la manera correcta de asignar los valores de una estructura a otra es la siguiente:

```
yo.edad = tu.edad
yo.altura = tu.altura
```

Listas de Estructuras en Python

Para declarar un arreglo de estructuras se hace de manera similar que la declaración de listas, pero ahora sus elementos son instancias de un tipo estructurado. Por ejemplo, si se quiere crear un arreglo de tres personas se escribiría

```
tresPersonas = [Persona(),Persona(),Persona()]
```

Para usar inicializar los valores de esta lista de estructuras debe acceder individualmente cada posición de la lista y dentro de ella cada campo, como se observa a continuación

```
tresPersonas[0].edad = 15
```

```
tresPersonas[0].altura = 1.65
tresPersonas[1].edad = 25
tresPersonas[1].altura = 1.75
tresPersonas[2].edad = 35
tresPersonas[2].altura = 1.55
```

Si se quiere realizar un cálculo sobre el arreglo de estructuras, por ejemplo, calcular el promedio de edades, se puede usar un for de la siguiente manera

```
suma = 0
for i in range(0,3):
  suma = suma + tresPersonas[i].edad
promedio = suma / 3
```

ADVERTENCIA: De igual manera, que con las variables declaradas como estructuras, no es correcto asignar directamente las posiciones de un arreglo de estructuras, pues también genera efectos de borde. Es necesario asignar cada campo individualmente. Por ejemplo:

```
tresPersonas[0] = trespersonas[2] # Prohibido, es incorrecto
```

La manera correcta es

```
tresPersonas[0].edad = tresPersonas[2].edad
tresPersonas[0].altura = tresPersonas[2].edad
```

Verificación de las aserciones correspondientes a las precondiciones y postcondiciones

La verificación de la corrección de un programa sólo puede hacerse por métodos de pruebas formales tal como se enseña en el curso teórico “Algoritmos y Estructuras 1” [1]. Hay estrategias para el diseño de programas basados en las reglas de estas pruebas que permiten derivar programas correctos a partir de sus especificaciones formales. La limitación de dichas estrategias es que existen problemas cuyas soluciones algorítmicas no pueden derivarse con estas técnicas.

En la práctica profesional lo que se desea es el poder escribir programas que cumplan cabalmente con sus precondiciones y postcondiciones, a esto se le llama “programación basada en contrato”. Se desea asimismo que los programas sean escritos de forma tal que se eviten condiciones de error que de antemano se sabe que pueden ocurrir durante la ejecución, a esto se le llama "programación robusta". El propósito del presente prelaboratorio es aprender a hacer programación basada en contrato y programación robusta.

Esto se logra añadiendo al código del programa acciones que verifiquen el cumplimiento de las aserciones de corrección durante la ejecución (mediante la instrucción "try/except"). En caso de violación de alguna aserción, la acción a tomar es dar un mensaje informativo al usuario del error ocurrido y parar la ejecución del programa (mediante la instrucción sys.exit()), de manera que no vayan a propagarse el error.

En el caso que el error sea por el incumplimiento de alguna especificación de entrada, la programación robusta indica que, de ser posible, en lugar de parar la ejecución del programa, se le da al usuario la opción de corregir el valor de entrada erróneo (se sugiere usar la instrucción "while").

Try/except

En el caso de Python, utilizaremos la instrucción `try/except` para tomar el control luego que un aserción falle. La idea básicamente es colocar las aserciones dentro de un bloque try/except, y en caso de fallar, el programa automáticamente comenzará a ejecutar las instrucciones indicadas en el except. La sintaxis es de la siguiente forma:

```
try:
 a = -1 # se coloca esta asignación a propósito a manera de
        # ejemplo
 assert( a >= 0 )
except:
 print("El numero no es positivo")
```

El código mostrado imprimirá el siguiente mensaje: “`El numero no es positivo`”.

A continuación se coloca un ejemplo utilizando `sys.exit()`. Para utilizar `sys.exit()` es necesario importar la librería `sys` usando la instrucción `import` al inicio del programa, de la siguiente forma:

```
import sys

try:
 a = int(input(“Introduzca un numero entero: ”))
 # el usuario coloca 1.5
 assert( a >= 0 )
except:
 print("El numero no es valido, debe ser un entero positivo")
 print("El programa terminara")
```

```
      sys.exit()
```

En este caso, cuando el usuario no introduce un valor apropiado, el programa termina, de manera similar a como lo hace el `assert`, sólo que se puede dar un mensaje más entendible para el usuario. A continuación se coloca un ejemplo de programación robusta, donde el usuario tiene la posibilidad de volver a introducir el valor.

```
while True:
   try:
     a = int(input("Coloque un entero positivo: "))
     assert( a > 0 )
     break
   except:
     print("Hubo un error en los datos de entrada")
    print("Vuelva a intentar")
```

La instrucción **`break`** se utiliza para salir del `while`, es decir, el programa continúa su ejecución luego del bloque de instrucciones internas al `while`.

En el caso de las postcondiciones, se puede hacer un tratamiento similar para indicarle al usuario de manera más "amigable" que el programa no logró verificar la postcondición, indicando los valores actuales de las variables involucradas en dicha expresión. En ese caso tiene más sentido que el programa termine la ejecución porque luego de la postcondición sólo queda la salida de los resultado. Veamos un ejemplo:

```
try:
 assert( r == (suma <= x + y))
except:
 print("Hubo un error en los calculos")
 print("La expresion r == (suma <= x + y) no es correcta")
 print("r="+str(r)+" suma="+str(suma)+" x="+str(x)+" y="+str(y))
 sys.exit()
```

Verificación de las aserciones correspondientes a los invariantes y función de cota de una iteración

La verificación de la corrección de un ciclo requiere conocer el invariante que se satisface durante toda la ejecución del mismo, así como la función de cota que garantiza que dicho ciclo termina. Las pruebas formales que garantizan que un ciclo es correcto se estudian en el curso teórico "Algoritmos y Estructuras 1" [1]. Usualmente el invariante involucra obtener el

valor de un cuantificador o de una función de agregación. Hasta ahora lo hemos verificado usando la instrucción `assert`.

Para encontrar errores en invariantes o función de cota planteada de manera práctica, se puede usar como estrategia la visualización, en cada iteración, de las variables que intervienen en el cuerpo del ciclo. Aquí también es útil el uso de la instrucción `try/except`. Asimismo, podemos verificar a través de una instrucción `try/except` si la función de cota es mayor o igual que cero y si es estrictamente decreciente. Observe que en caso de usar un ciclo `FOR`, esto no es necesario pues el `FOR` de Python siempre termina.

El propósito del presente prelaboratorio es aprender a verificar que las instrucciones iterativas son robustas mediante la inclusión en el código de acciones que permiten visualizar los valores de las variables y el chequeo de la finitud del ciclo, de manera “amigable” para el usuario.

En caso de violación de alguna de las aserciones correspondientes a invariantes o cotas, la acción a tomar es dar un mensaje informativo al usuario del error ocurrido y parar la ejecución del programa (mediante la instrucción `sys.exit()`), de manera tal que se garantice que la ejecución del ciclo es correcta y que no se cae en un ciclo infinito. Note que esto garantiza que los valores de las variables involucradas en el ciclo son correctos, así como el valor de la cota, a menos que ocurra una violación de un `assert`.

Veamos un ejemplo estudiado en el prelaboratorio 3. El siguiente programa calcula la suma de los números pares entre 0 y N. El programa tiene las aserciones correspondientes al invariante y la cota. Se incluye la verificación usando `try/except`.

```
k,suma=0,0
cota = N-k+1

# Verificacion de invariante al inicio
try:
  assert( 0<=k<=N and suma == sum ( x for x in range(0,k) if (x % 2 ==0) ) )
except:
  print("Hubo un error en el invariante para los siguientes valores")
  print("k="+str(k)+" suma="+str(suma))
  sys.exit()

# Verificacion de cota al inicio
try:
  assert( cota >= 0 )
except:
  print("Error cota no positiva: ")
  print("cota="+str(cota))
  sys.exit()

while ( k < N ):
```

```
if (k % 2 == 0):
   suma=suma+k
else:
   pass
k=k+1

# Verificacion de invariante en cada iteracion
try:
   assert( 0<=k<=N and suma == sum ( x for x in range(0,k) if (x % 2 ==0) ) )
except:
   print("Hubo un error en el invariante para los siguientes valores")
   print("k="+str(k)+" suma="+str(suma))
   sys.exit()

# Verificacion de cota decreciente en cada iteración
try:
   assert( cota > N-k+1 )
except:
   print("Error cota no decreciente : ")
   print("cota anterior ="+str(cota)+" nueva cota ="+str(N-k+1))
   sys.exit()

cota = N-k+1
# Verificacion de cota positiva en cada iteración
try:
   assert( cota >= 0 )
except:
   print("Error cota no positiva: ")
   print("cota="+str(cota))
   sys.exit()
```

Análisis descendente y Subprogramas no recursivos.

En los ejercicios que se han desarrollado a lo largo del curso, se puede observar una diferenciación natural de secciones dentro del bloque de instrucciones que conforma el código del programa. En casi todos ellos hay una sección de adquisición de datos (entrada), una sección de procesamiento algorítmico (cálculos) y una sección de despliegue de los resultados (salida).

Cuando la complejidad de un programa aumenta, es conveniente separar estas secciones en estructuras de programación que se conocen como subprogramas. Estos subprogramas pueden a la vez ser de cierta complejidad por lo que se requiere subdividirlos para poder resolver el problema de manera efectiva y legible. A esto se le conoce como la estrategia de “Divide and Conquer” que puede traducirse como “Divide y Vencerás”.

Al proceso de concebir la solución de un problema con esta estrategia se le conoce como “Análisis Descendente”. Este mismo término (análisis descendente) se usa para designar

al diseño resultante de este proceso. El análisis descendente de un problema puede plasmarse en un diagrama llamado “Carta Estructurada”. Éste ayuda en la comprensión y el desarrollo del programa.

Una ventaja de tener subprogramas es que ellos pueden ser reutilizados en distintas partes del código. El lenguaje de programación Python provee una estructura de subprograma, conocida como función. Con esta estructura se pueden traducir los procedimientos y las funciones de GCL usados en el curso de Algoritmos y Estructuras 1.

La estructura general de una función en Python se inicia con la palabra reservada `def`, de manera similar a como se definen los predicados que habíamos visto en los laboratorios previos, seguida del nombre de la función con la lista de parámetros colocados entre paréntesis. Cada parámetro puede ser seguido de dos puntos (`:`), y el tipo de variable que se espera obtener. Para especificar el tipo de valor de retorno, coloque la símbolo `->` y luego el nombre del tipo. A continuación se coloca un ejemplo:

```
def perimetro(r: float, pi: float) -> float:
# PRECONDICION: r>0
# POSTCONDICION: perimetro = 2*pi*r

# var respuesta: float // variable auxiliar

  respuesta = 2 * pi * r

  return respuesta
```

Para realizar la llamada a una función, se coloca el nombre de ésta seguido de los parámetro reales separados por coma y encerrados entre paréntesis. Esta llamada debe colocarse dentro de una expresión del mismo tipo del valor de retorno. Por ejemplo, en una asignación a una variable, como se muestra a continuación:

```
# var a: float
a = perimetro( 20.5, 3.141 )
```

En el caso de los procedimientos que se pueden definir en GCL, también pueden traducirse a Python como funciones. Indicando que no tienen valor de retorno. Para esto se utiliza la palabra reservada `'void'`. En este caso, en la especificación del tipo de cada uno de los parámetros, se puede colocar un string con la palabra ‘void’. Por ejemplo:

```
def inicializarLista (l:[int]) -> 'void':
# PRECONDICION: true
```

```
# POSTCONDICION: (all l[i]==0 for i in range(0,len(l))

# var i: int
 for i in range(0,len(l)):
    # invariante:
    assert(all l[k]==0 for k in range(0,i)
    l[i] = 0
```

El lenguaje Python permite el uso recursivo de subprogramas, esto es que un subprograma tenga una llamada a sí mismo de forma directa o indirecta. Esto es algo de mayor complejidad al requerido en el taller de esta semana, pues será cubierto más adelante en este curso y en los cursos más avanzados. En este laboratorio sólo trabajaremos con subprogramas no recursivos. Igual que en los programas, los subprogramas debe tener especificadas las precondiciones y las postcondiciones, como se pudo observar en los ejemplos dados. Para más información en cuanto a funciones en python, se puede revisar la referencia [1].

Ejercicio: Tome el ejercicio 2 realizado en el prelaboratorio 5 (Prelab5ejercicio2.py), que calcula las raíces de una ecuación cuadrática. Escriba un subprograma que realice dicho cálculo. Modifique el programa principal para que realice la llamada a este subprograma. Recuerde que cada función debe tener su precondición y postcondición.

Pasaje de parámetros por valor y por referencia.

Una diferencia notable entre los procedimientos y funciones del lenguaje de programación formal GCL, es la forma de pasaje de parámetros. En GCL se distinguen parámetros de entrada, parámetros de salida y parámetros de entrada y salida. El estudio de éstos es parte del contenido del curso teórico de Algoritmos y Estructuras I.

Los parámetros por valor son aquellos donde el valor del parámetro real es copiado al parámetro formal. Los parámetros por referencia son aquellos donde una dirección o referencia del parámetro real es copiada en el parámetro formal. Los parámetros por valor corresponden a los parámetros de entrada de GCL, mientras que los parámetros por referencia corresponden a los parámetros de entrada y salida de GCL.

En Python, los parámetros que son de algún tipo básico del lenguaje (como int, char, float o str) son tratados como parámetros por valor. Aquellos que corresponden a tipos no básicos como arreglos o estructuras son tratados como parámetros por referencia. También se puede devolver una tupla con varios valores de resultado. Esto es similar a los parámetros de salida de GCL, que se usan cuando se quiere devolver más de un valor.

Veamos un ejemplo.

```
def f(a: str, b: int) -> (str, int):
# PRECONDICION: b>0
# POSTCONDICION: f = (‘valor nuevo’,b+1)
  a = ‘valor nuevo’
  b = b + 1

  return a, b
```

Pruebe las siguientes acciones en el interpretador de Python.

```
x, y = 'valor viejo', 99
w, z = f (x, y)
print(x, y)
print(w, z)
```

En el caso de arreglos o estructuras que son tratados como parámetros por referencia, si se modifica dentro del procedimiento algún valor interno de dicha estructura (alguna posición del arreglo o algún campo de la estructura) esta modificación queda reflejada al salir del procedimiento, es decir, en el programa principal se recibe dicha modificación. Pero si se modifica la referencia, por ejemplo al tratar los arreglos por concatenación (ejemplo A = A+[r]) esta modificación no se verá reflejada porque la referencia de dicho arreglo es tratada por valor es decir no puede ser cambiada. Por ello, debe evitarse este tipo de asignaciones dentro de funciones. Se recomienda modificar los arreglos usando su índices.

Existen diferentes tratamientos para el pasaje de parámetros, pero serán vistas en cursos más avanzados.

Alcance de identificadores.

Alcance es la palabra que utilizamos para describir la habilidad para acceder a una variable desde algún lugar de nuestro código.

Hasta ahora hemos trabajado con dos tipos de alcances, los alcances globales y los alcances en funciones. Expliquemos los dos tipos de alcances de variables con el siguiente ejemplo:

```
globalVar = "Esta es de alcance global"
def miFuncion() -> ‘void’:
 localVar = "Esta es local"
 print("miFuncion - localVar:  " + localVar)
 print("miFuncion - globalVar: " + globalVar)
miFuncion()
print("global - globalVar: " + globalVar)
print("global - localVar:  " + localVar)
```

Coloque el ejemplo en un archivo, y corra el programa. Observe lo que ocurre. Note que el programa falla porque el identificador `localVar`, está definido sólo dentro del subprograma `miFuncion`. Mientras que el identificador `globalVar` está definido en todo el programa.

Dentro de la declaración de un subprograma puede usarse un identificador que ya haya sido usado en el programa, con una definición nueva (diferente). Esto sobrescribe la definición a efectos del subprograma, hasta el final de su bloque de instrucciones. Los identificadores declarados dentro de un subprograma se dice que tiene alcance local.

En el cuerpo de un subprograma se pueden usar los identificadores definidos dentro del subprograma, es decir los identificadores locales. También se pueden usar los identificadores definidos antes de la declaración del subprograma que no hayan sido redefinidos dentro del mismo. En el ejemplo anterior esto último es el caso de `globalVar`. A estos últimos identificadores se les dice que son identificadores que designan elementos del "contexto global" del subprograma (o simplemente elementos o identificadores globales).

El uso de variables y parámetros globales tiene serios inconvenientes para la legibilidad de los programas y para la verificación de su corrección. Es aconsejable que se pase por parámetro cualquier dato que quiera compartirse entre subprogramas o entre el programa y un subprograma.

De manera que en este curso estará prohibido el uso de variables globales. Esto es, aunque las reglas de alcance permiten a un subprograma usar una variable o un parámetro de su contexto global, vamos a exigir que en el bloque de instrucciones se use sólo parámetros y variables declarados en el contexto local del propio subprograma.

Principios de los procesos de desarrollo ágil.

El desarrollo ágil es muy utilizado actualmente para crear sistemas de software. En éste participan pequeños equipos que trabajan en colaboración organizada y disciplinada. La idea es desarrollar software en lapsos cortos. Se basa en ciertos principios, entre los cuales destacan:

- El software que funciona es la medida principal de progreso.
- Los procesos ágiles promueven un desarrollo sostenido, es decir, el equipo debe mantener un ritmo constante de forma indefinida.
- La atención continua a la calidad técnica y al buen diseño mejora la agilidad.
- La simplicidad es esencial.

El desarrollo ágil promueve ciertas prácticas que son esenciales para obtener software de calidad. El mayor beneficio de las prácticas se consigue con su aplicación conjunta y equilibrada puesto que se apoyan unas en otras. Algunas de ellas que podemos aplicar en un primer curso de programación son:

- Diseño simple: Se debe diseñar la solución más simple que pueda funcionar y ser implementada en un momento determinado del proyecto.

- Refactorización (Refactoring): Es una actividad constante de reestructuración del código con el objetivo de remover duplicación de código, mejorar su legibilidad, simplificarlo y hacerlo más flexible para facilitar los posteriores cambios. Se mejora la estructura interna del código sin alterar su comportamiento externo.
- Programación en parejas: Toda la producción de código debe realizarse con trabajo en parejas de programadores. Esto conlleva ventajas implícitas (menor tasa de errores, mejor diseño, mayor satisfacción de los programadores, entre otras).
- Propiedad colectiva del código: Cualquier miembro del equipo puede cambiar cualquier parte del código en cualquier momento.
- Estándares de programación: Se enfatiza que la comunicación de los programadores es a través del código, con lo cual es indispensable que se sigan ciertos estándares de programación para mantener el código legible.

Principios de buena programación.

Para adiestrarse en las prácticas de la programación ágil es imprescindible seguir los principios que rigen una buena programación. A continuación se listan aquellos que pueden ser aprovechados en un primer curso de programación.

- Evitar la repetición de código o documentación y en su lugar reutilizar.
- Principio de abstracción. “Cada pieza de funcionalidad significativa en un programa debe ser implementada en un sólo lugar del código fuente”
- La mejor solución a un problema es la más simple. Ejemplo:

```
 # EJEMPLO DE UNA SOLUCIÓN POCO SIMPLE

 i=0
j=100
while ( i < j ):
    # Este ciclo tiene 50 iteraciones.
    i = i+1
    j = j-1


# EJEMPLO DE UNA SOLUCIÓN SIMPLE

i = 0
while ( i < 50 ):
    # Este ciclo también tiene 50 iteraciones.
    i = i + 1
```

- Hacer pruebas unitarias.

- Usar identificadores descriptivos. Los nombres de las funciones, variables, etc. deben declarar claramente lo que hacen. Ejemplos:

```
# EJEMPLO DE NOMBRES POCO DESCRIPTIVOS
dia0 = int( input( "Dia de nacimiento: " ) )
mes0 = int( input( "Mes de nacimiento: " ) )
año = int( input( "Año de nacimiento: " ) )
dia1 = int( input( "Dia actual: " ) )
mes1 = int( input( "Mes actual: " ) )
año1 = int( input( "Año actual: " ) )
```

```
# EJEMPLO DE NOMBRES DESCRIPTIVOS
diaDeNacimiento = int( input( "Dia de nacimiento: " ) )
mesDeNacimiento = int( input( "Mes de nacimiento: " ) )
añoDeNacimiento = int( input( "Año de nacimiento: " ) )
diaActual = int( input( "Dia actual: " ) )
mesActual = int( input( "Mes actual: " ) )
añoActual = int( input( "Año actual: " ) )
```

- Principio del menor asombro: El nombre de un subprograma debería corresponder con lo que hace.
- Principio de responsabilidad única. Un componente de código (subprograma) debe ejecutar una única y bien definida tarea.
- Minimizar el acoplamiento. Cada componente (bloque de código, subprograma, etc.) debe minimizar las dependencias de otros componentes.
- Maximizar cohesión. Evitar implementar en un componente dos funcionalidades que no están relacionadas, cumpliendo tareas que no tienen relación.
- La reutilización de código es buena.
- No usar más parámetros de entrada de los que se necesitan. Asegurarse de que los parámetros recibidos son los que se esperan.

Programación en Pareja.

La Programación por Parejas es considerada como uno de los pilares de una metodología ágil. Es una técnica relativamente sencilla de aplicar en la cual dos programadores trabajan codo con codo, en un único computador, para desarrollar el código de un programa. En este contexto, la persona que teclea recibe el nombre de conductor, mientras que la persona que revisa el código recibe el nombre de observador o navegante. Estos dos roles, conductor y observador/navegante, en ningún momento, deben ser considerados como roles estáticos.

Los miembros de la pareja deben intercambiar sus roles de manera frecuente durante toda la actividad de desarrollo. Si no se cumple este sencillo requisito, no se podrá hablar de una verdadera Programación por Parejas. El objetivo de la Programación por Parejas es mejorar la calidad del código. Las buenas prácticas en la programación por parejas son:

- Todo se comparte: Los dos miembros de la pareja, sin importar el rol que estén desempeñando en un determinado momento, son los autores del código. Ambos tendrán que aprender a compartir sus éxitos y sus fracasos, expresándose mediante frases como "cometimos un error en esta parte del código" o, mucho mejor, "hemos pasado todas las pruebas sin errores".
- Observador activo: No debe caerse en el error de confundir el rol de observador con disponer de un periodo de descanso, realizar una llamada o contestar a un mensaje de texto recibido en el teléfono. Por el contrario el observador activo ocupa su tiempo realizando un proceso continuo de análisis, diseño y verificación del código desarrollado por el conductor, tratando en todo momento de que la pareja sea capaz de alcanzar la excelencia del código.
- Deslizar el teclado, no cambiar las sillas: Esta regla nos indica que el espacio de trabajo debe estar dispuesto de modo que no sea necesario intercambiar el lugar frente al teclado (cambiar las sillas) para poder intercambiar los roles. Con el simple movimiento de deslizar el teclado para que éste cambie de manos la pareja debe ser capaz de intercambiar los roles de conductor a observador/navegante y viceversa.
- La importancia del descanso: Deben realizarse descansos de manera periódica, que ayudarán a la pareja a retomar el desarrollo con la energía necesaria, por lo que es imprescindible que,durante los mismos, los dos desarrolladores mantengan sus manos y su mente alejados de la tarea que estén desarrollando en pareja. Algunas tareas que se pueden realizar durante estos periodos de descanso son: revisar el correo electrónico personal, realizar una llamada de teléfono pendiente, contestar a los mensajes de texto recibidos en el móvil, navegar por la red, comer,incluso, estudiar otra materia.

Programación Sostenida.

Es importante llevar un ritmo sostenido de trabajo. El concepto que se desea establecer con esta práctica es el de planificar el trabajo para mantener un ritmo constante y razonable, sin sobrecargar al equipo. Cuando un proyecto se retrasa, trabajar tiempo extra puede ser más perjudicial que beneficioso. El trabajo extra desmotiva inmediatamente al grupo e impacta en la calidad del producto.

Programación dirigida por las pruebas (“Test-driven programming”)

En las metodologías tradicionales, la fase de pruebas, incluyendo la definición de los tests, es usualmente realizada sobre el final del proyecto, o sobre el final del desarrollo de cada módulo. Las metodologías ágiles proponen un modelo inverso, en el que, lo primero que se escribe son las pruebas que el sistema debe pasar. Luego, el desarrollo debe ser el mínimo necesario para pasar las pruebas previamente definidas. Estas pruebas, llamadas pruebas unitarias, corresponden a probar cada componente de software (o subprograma) de manera individual. Para ello, hay que intentar que los casos de prueba elegidos cubran la posibilidad de encontrar el mayor número de errores posibles, ya que los errores permanecerán ocultos hasta que se ejecute la porción de código que los provoca y para que dicha porción se ejecute es necesario que el programador realice las acciones necesarias para provocarlo.

Algunas recomendaciones para hacer pruebas: verificar el tipo de datos de entrada, las precondiciones, los resultados esperados (correctos o incorrectos). En resumen se está creando los “escenarios de pruebas”, es decir, las diferentes condiciones en las que el programa deberá trabajar.

Lectura y escritura en archivos

En Python se pueden manejar archivos de texto, es decir su contenido se interpreta como caracteres por lo que se leen y escriben strings desde y hacia el archivo. Las principales operaciones son:

- **OPEN**

open() devuelve un objeto tipo archivo, y es comúnmente usado con dos argumentos, con la siguiente sintaxis

```
open(nombre_de_archivo, modo)
```

El primer argumento `nombre_de_archivo` es un string que contiene el nombre/dirección del archivo. El segundo argumento es otro string que contiene un par de caracteres que describen la forma o `modo` en que se utilizará el archivo. El `modo` puede ser `'r'` cuando sólo se necesita leer el archivo, `'w'` para sólo escritura (si existe un archivo con el mismo nombre será borrado o reescrito), y `'a'` que abre el fichero para añadir nuevos caracteres; por lo que todos los datos escritos en el archivo se añade automáticamente al final. También, está el modo `'r+'` que abre el archivo para lectura y escritura. El argumento de modo es opcional por lo que se supondrá `'r'` si se omite. Por ejemplo,

```
f = open('workfile', 'w')
```

abre el archivo para escritura.

Al hacer la apertura de un archivo se genera un objeto de tipo archivo (`f`) que permite realizar ciertas operaciones las cuales se explicarán más adelante.

En el modo de texto, el valor por defecto en la lectura para fin de línea depende de la plataforma específica (`\n` en Unix, `\r\n` en Windows) convirtiéndolas simplemente en `\n` . En la escritura, el valor por defecto para fin de línea se convierte en el caracter específico según la plataforma.

- **REMOVE**

En caso de querer eliminar un archivo, puede utilizar la función `os.remove(string)`, siendo string la dirección del archivo. Para utilizar estas funciones, debe importar la librería OS. usando la instrucción: `import os.`

Operaciones con Archivos

Como se vió en la sección anterior, al abrir un archivo se crea un objeto de tipo archivo al cual denominamos f.

Si se quiere leer el contenido de un archivo, se utiliza la función `f.read(tamaño)`, la cual lee la cantidad de datos indicado en `tamaño`. El argumento `tamaño` es opcional, y en caso de omitirse, la función devolverá el archivo completo como un string. Si la función llega a fin del archivo, devuelve una cadena vacía.

También se puede utilizar la función `f.readline()` para leer una línea del archivo. Si el `readline` devuelve una cadena vacía significa que se ha llegado al fin de archivo.

Se puede leer líneas de un archivo de forma iterativa usando un `for`, de una manera sencilla como se muestra a continuación.

```
for line in f:
     print(line, end='')
```

Si se desean leer todas las líneas de un archivo y colocarlas en una lista, puede utilizar `list(f)` o `f.readlines()`. Por ejemplo, imaginemos que se tiene un archivo de texto, con el nombre "`input.txt`". El archivo contiene el siguiente texto:

```
Linea 1
Linea 2
```

Si se desea leerlo con la operación `f.readlines()`, se puede hacer la siguiente secuencia de instrucciones:

```
f = open("input.txt")
lineas = f.readlines()
```

El resultado es una lista formada por las líneas del archivo:

```
['Linea 1\n', 'Linea 2\n']
```

La variable "`lineas`" será una lista con los dos strings: "`Linea 1\n`", y "`Linea 2\n`".

Para escribir contenido en un archivo, se puede utilizar la operación `f.write(frase)`, siendo `frase` la cadena a escribir. La operación devuelve la cantidad de caracteres escritos. Por ejemplo:

```
>>> f.write('Hello World')
11
```

Para escribir en el archivo, algo diferente a un string, primero debe convertirse a string. Veamos un ejemplo:

```
>>> valor = [2, 3, 4]
>>> s = str(valor)
>>> f.write(s)
9
```

Luego de haber leído el archivo, es necesario llamar a la operación `f.close()`. Esto cerrará y liberará los recursos utilizados para mantener el archivo abierto. También se tiene la función `f.closed` que dice si el objeto archivo `f` está cerrado.

Como buena práctica al trabajar con archivos, es utilizar un bloque de instrucciones que comienza con la palabra clave `with`. Dicho bloque termina con la operación `f.closed`. Esto garantiza que el archivo sea cerrado correctamente a pesar de haber ocurrido algún error en el camino. A continuación se coloca un ejemplo:

```
>>> with open('nombreDeArchivo', 'r') as f:
...   read_data = f.read()
>>> f.closed
```

En este caso, en el bloque dentro del `with`, se mantendrá el archivo abierto. Al salir del bloque, existe la garantía de que estará cerrado. En las 3 líneas de código del ejemplo anterior, se abrió el archivo, y se leyeron todos los datos con la operación `f.read()`. Al final, la operación `f.closed` confirma que está cerrado. En la variable `read_data` se almacenan todos los datos leídos del archivo, pues es la acción por defecto, cuando no se especifica el tamaño a leer.

Recursión

Como se ha explicado en los laboratorios anteriores, un problema complicado puede ser dividido en uno o más subproblemas. Realizar tal división corresponde a hacer un *análisis descendente* del problema en cuestión. Cuando el análisis descendente de un problema incluye la resolución de otra instancia de ese mismo problema, diremos que el análisis propone una solución *recursiva.* Muchos problemas pueden resolverse usando recursión. En particular, si un problema puede dividirse en una o más instancias del mismo problema, pero con una entrada más pequeña, diremos que la solución propuesta utiliza la técnica: *divide y conquistaras.*

.

Recurrencias.

Muchas funciones y propiedades son enunciadas en formas de *recurrencias*. Por ejemplo, a continuación se muestra la recurrencia que define la serie de Fibonacci.

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Soluciones Recursivas.

La recursión no sólo puede utilizarse para calcular el valor de funciones, sino también para resolver problemas. En particular, la resolución de un problema puede depender de la resolución de instancias más pequeñas de ese problema.

Por ejemplo:

- Consideremos que tenemos un arreglo de enteros, ordenados de menor a mayor.
- Queremos decidir si un elemento **elem** se encuentra en el arreglo o no.

Una posible solución sería recorrer el arreglo y comparar cada elemento con **elem.** Sin embargo, podemos formular una solución alternativa:

- Se revisa un elemento aleatorio **x** del arreglo y se compara con **elem**
- Si **x == elem** entonces reportamos que el elemento se encuentra en el arreglo.
- De lo contrario, pudieran ocurrir dos casos:

- Si **x < elem** debemos reformular la búsqueda. Sin embargo, cualquier elemento anterior a **x** es menor o igual que **x** (ya que supusimos que el arreglo estaba ordenado). Por lo tanto, cualquier elemento anterior a **x** está estrictamente menor que **elem.** Siendo así,

podemos restringir nuestra búsqueda solamente a los elementos que estén después de **x.**

- Si **x > elem** podemos utilizar un argumento análogo al anterior, para justificar que la búsqueda se realice únicamente sobre los elementos que estén antes de **x.**

Notemos que el arreglo que se pasa como entrada a los subproblemas es siempre más pequeño que el original. Eventualmente, el arreglo será de tamaño cero. En tal caso, ya no tiene sentido dividir el problema. Podemos simplemente reportar que el elemento no se encuentra en el arreglo (no hay elemento alguno en un arreglo vacío).

Podemos formular nuestra solución recursiva como una recurrencia, de la siguiente manera:

$$esta(A, elem, i, j) = \begin{cases} false & i >= j \\ true & A[i] = elem \\ esta(A, elem, k+1, j) & A[i] < elem, i <= k < j \\ esta(A, elem, i, k) & A[i] > elem, i <= k < j \end{cases}$$

Nótese que se toma una k arbitraria, tal que i ≤ k < j . Resultado en investigación han demostrado que, en la mayoría de los casos, la mejor escogencia para tal k es el valor (i+j)/2. Esto implica que el elemento a comparar del arreglo, siempre será aquel que se encuentre en la mitad de la porción considerada del mismo. A la estrategia anterior, junto con la escogencia propuesta para k, se le conoce como el algoritmo de *búsqueda binaria*.

Este algoritmo es mucho más eficiente que la propuesta original de recorrer el arreglo completo. La razón por la que es tan eficiente será discutida más adelante, en el curso de Algoritmos y Estructura II

Recursión Indirecta

Es posible que un problema tenga una solución que sea recursiva, pero de manera indirecta. Por ejemplo, un subprograma *foo* que llame a otro subprograma *bar*, cuando a su vez *bar* llama a *foo*. Tal clase de recursión es conocida como *recursión indirecta*.

Existen muchos ejemplos concretos de recursión indirecta que se tratarán en cursos futuros. Sin embargo, a efectos de este laboratorio solo trataremos con subprogramas directamente recursivos. Esto no nos restringe en poder de expresión, ya que cualquier conjunto de subprogramas que sea indirectamente recursivo, puede re-escribirse como un solo subprograma directamente recursivo.

Iteración vs Recursión.

Una propiedad fundamental en la computación es que todo problema que tenga una solución iterativa, también tendrá una recursiva. Así mismo, todo problema que tenga una solución recursiva, también tendrá una iterativa. Escoger qué solución implementar dependerá de muchos factores, como la eficiencia, la facilidad de codificación, la legibilidad, entre otros.

Eficiencia de la Iteración.

Generalmente, una solución iterativa será más eficiente que una recursiva (la razón de esto será mucho más clara adelante, en el curso de Organización y Arquitectura del Computador). Una solución iterativa no debe invocarse a sí misma, lo cual tendría un costo en tiempo y espacio. Sin embargo, diseñar soluciones iterativas no siempre es sencillo. Muchos problemas tienen soluciones que son naturalmente recursivas. Un buen ejemplo es el cálculo de la serie de Fibonacci. La solución recursiva es inmediata de la definición, sin embargo es sumamente ineficiente. La versión iterativa, aunque mucho más eficiente, es más difícil de diseñar.

Recursión de Cola.

La recursión es tan natural y común, que muchos lenguajes han adoptado una optimización especial conocida como *Recursión de Cola*. La optimización consta en transformar automáticamente un subprograma recursivo por una versión iterativa, siempre y cuando tal subprograma cumpla con una característica particular: cada llamada recursiva no debe ser sucedida por acción alguna. **Hay que tener en cuenta que Python no optimiza la recursión de cola.**

Consideremos la recurrencia siguiente, que define el factorial de un entero:

$$fact(n) = \begin{cases} 1 & n = 0 \\ n * fact(n-1) & n > 0 \end{cases}$$

Al implementar tal solución directamente, la llamada recursiva a fact sería sucedida por la multiplicación con n. Por lo tanto, tal implementación no sería recursiva de cola. Consideremos ahora, la recurrencia en (4), que define el factorial de una manera alternativa, usando un acumulador como ayuda:

$$fact(n) = fact\_aux(n, 1)$$

$$fact\_aux(n, acum) = \begin{cases} acum & n = 0 \\ fact(n-1, acum * n) & n > 0 \end{cases}$$

Si implementamos directamente esta nueva solución, podremos notar que la función recursiva fact_aux si es recursiva de cola. Una vez se realiza la llamada recursiva, no hay que realizar acción alguna adicional.

Terminación de una Recursión.

Así como en una iteración se debe plantear una cota (con la finalidad de asegurar que tal iteración termina), en un subprograma recursivo se debe plantear igualmente una cota. Esto, ya que se corre el mismo peligro de no terminar. Particularmente, si las llamadas recursivas no se hacen sobre instancias más pequeñas que la original, entonces el programa puede recurrir sin fin.

De ésta manera, todo subprograma recursivo debe incluir una función de cota que:

- Debe depender únicamente de los parámetros de entrada
- Debe ser siempre no-negativa
- En cada llamada recursiva, debe decrecer (la función de cota del subprograma llamado, debe ser estrictamente menor que la del llamador).

Referencias:

[1] Guía de estilo del código Python, 10 de Agosto de 2007, Disponible en la web: http://mundogeek.net/traducciones/guia-estilo-python.htm