Práctica 12: Recursividad

Estructuras de Datos y Algoritmos I

Autor: José Mauricio Matamoros de Maria y Campos

1. Objetivo

El alumno aprenderá a modelar problemas de forma recursiva a fin de utilizar el concepto de recursividad en la solución de problemas.

2. Material

Se asume que el alumno cuenta con un una computadora con arquitectura Intel x86 o compatible con interprete de python instalado (versión 3.5 o posterior).

2.1. Instalación de Python 3 en Ubuntu Linux

Para instalar Python3 en Ubuntu Linux ejecute el siguiente comando en la terminal (se requieren privilegios de superusuario).

```
sudo apt-get update
sudo apt-get -y install python3-all
```

O bien, si se desea instalar sólo los paquetes mínimos necesarios

```
sudo apt-get -y install python3 python3-pip python3-numpy python3-matplotlib
```

3. Antecedentes

La recursividad (o recursión) es una forma de modelar problemas de forma tal que la solución está basada en la propia definición del problema, es decir, la solución del problema está compuesta por la composición de las soluciones de variantes de menor grado u orden del mismo problema.

El ejemplo más simple de recursión es el cálculo del factorial de un número entero $n \in \mathbb{N}$ que se define como:

$$n! = f(n) = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \cdots \times n \tag{1}$$

Para calcular el factorial de un número de forma recursiva es necesario un cambio de paradigma y modelar el problema no de forma ascendente y sintética (construirlo desde 1 hasta n), sino de forma descendente y analítica: descomponiento la solución hasta llegar a un caso trivial. Es por esto que los modelos matemáticos recursivos siempre tienen al menos dos casos (caso recursivo y caso trivial). Esto es:

$$n! = f(n) = \begin{cases} n \cdot (n-1)! & \text{Si } n \geq 2 \quad (2a) \\ 1 & \text{Si } n < 2 \quad (2b) \end{cases}$$

De este modo, la Ecuación (2a) es la definición recursiva de la función factorial, mientras que Ecuación (2b) es la condición de parada de la recursividad o caso base.

Esta forma particular de modelar problemas es usada en muchos paradigmas algorítmicos como *divide y vencerás*, *programación dinámica*, y *resolución reversa (backtracking)*.

3.1. Búsqueda binaria

Una búsqueda consiste en localizar un elemento particular denominado *llave* dentro de un conjunto. Cuando el conjunto está representado como un arreglo (o, en general, una lista) la única forma de localizar una llave en el arreglo es recorrer todo el arreglo de principio a fin (búsqueda lineal), es decir, comparar la llave con todos los elementos del conjunto. Para cualquier conjunto arbitrario esta operación no puede realizarse en un tiempo menor a $O(n)$ pues en el peor caso la llave buscada coincidiría con el último elemento a comparar en el conjunto.

Surge entonces una pregunta: ¿podrá buscarse más rápido? En general para un conjunto arbitrario esto no es posible, pero si se altera la naturaleza del conjunto se puede acelerar la búsqueda; por ejemplo usando un arreglo ordenado y la estrategia *divide y vencerás*.

Como todos los algoritmos modelados bajo el paradigma *divide y vencerás*, la ***búsqueda binaria*** primero divide un arreglo grande en dos subarreglos más pequeños y luego opera los subarreglos de forma recursiva (o iterativa). Sin embargo, en lugar de trabajar en ambos subarreglos, descarta un subarreglo y continúa en el segundo subarreglo. La decisión de descartar un subarreglo require de una sola comparación.

Por lo tanto, la búsqueda binaria reduce el espacio de búsqueda (segmento del arreglo en el cual se buscará la llave) a la mitad en cada paso, lo cual es posible sólo debido a que el arreglo está ordenado. Inicialmente, el espacio de búsqueda es el A arreglo completo y la búsqueda se realiza comparando a la llave k con el elemento en la posición $i = \frac{n}{2}$. Esto conduce a 3 casos:

- Si $k = A\left[\frac{n}{2}\right]$ la llave k se encuentra en el arreglo y, típicamente, se devuelve su posición.
- Si $k < A\left[\frac{n}{2}\right]$ se repite la búsqueda de la llave k en el subarreglo izquierdo $A_0^{\frac{n}{2}-1}$
- Si $k > A\left[\frac{n}{2}\right]$ se repite la búsqueda de la llave k en el subarreglo derecho $A_{\frac{n}{2}+1}^{n}$

La búsqueda se repite acortando el subarreglo de forma recursiva hasta que la llave k se encuentra o el sub-arreglo es vacío. En este último caso la llave no fue encontrada.

El algoritmo de búsqueda binaria es como sigue:

Algoritmo 1 Búsqueda binaria recursiva

```
procedure BINARY-SEARCH(key, A, i, d)      ▷ Donde key es la llave buscada,
                                                A es el arreglo donde se busca la llave key,
                                                i es el índice izquierdo que delimita el subarreglo A_i^d y
                                                d es el índice derecho que delimita el subarreglo A_i^d.
    if i > j then
        return -1                                                    ▷ No encontrado
    else                                                             ▷ Búsqueda recursiva
        mid ← i + ⌊(d−i)/2⌋
        if A[mid] == key then
            return mid                                               ▷ Llave encontrada
        else if A[mid] < key then
            return BINARY-SEARCH(key, A, i, mid − 1)                 ▷ Búsqueda en ramal izquierdo
        else
            return BINARY-SEARCH(key, A, mid + 1, d)                 ▷ Búsqueda en ramal derecho
        end if
    end if
end procedure
```

En cada salto el espacio de búsqueda se reduce a la mitad, es decir $n \to \frac{n}{2} \to \frac{n}{4} \to \frac{n}{8} \to \cdots \to 1$. Llamemos h al número de pasos tras el cual el espacio de búsqueda se agota y como la sucesión $n, \frac{n}{2}, \ldots, 1$ converge como $\frac{n}{2^h} = 1$, entonces $h = \log_2 n$. Por lo tanto, el tiempo requerido para buscar una llave en un conjunto ordenado es de tan solo $O(\log n)$.

3.2. El problema de las torres de Hanoi

Las Torres de Hanoi es un rompecabezas matemático que consiste en mover una pirámide de n discos de uno de los tres postes a cualquiera de los otros dos postes disponibles siguiendo un conjunto sencillo de reglas:

- Solo se puede mover un disco a la vez.
- Cada movimiento consiste en tomar el disco superior de una de las pilas y colocarlo encima de otra pila, es decir, un disco sólo se puede mover si está encima de una pila.
- No se puede colocar ningún disco encima de un disco más pequeño.

El rompecabezas comienza con los discos en una pila ordenada en orden ascendente en uno de los postes formando una pirámide cónica, cuidando que el disco más pequeño esté en la parte superior. El objetivo del rompecabezas es mover toda la pirámide a otro poste respetando las reglas.

El número mínimo de movimientos necesarios para resolver un rompecabezas de las Torres de Hanoi es $2^n - 1$, donde n es el número total de discos. El priblema original, según una de las leyendas de la India, habla de una habitación grande con tres postes y 64 discos de oro que un grupo de monjes Brahmin tendrá que mover y, cuando la Torre de Brahma haya terminado de moverse, el mundo llegará a su fin. Matemáticamente hablando, si se realizada un movimiento por segundo se necesitarían $2^{64} - 1 = 1.8410^{19}$ segundos para resolver el rompecabezas, es decir unos 585 mil millones de años, unas 41 veces la edad estimada del universo.

Implementación

Es posible resolver el rompecabezas usando estrategias recursivas. La llave para resolver el problema es romperlo en un conjunto de problemas más pequeños de forma sucesiva hasta dar con la solución.

Por ejemplo, el aloritmo propuesto para mover n discos del poste p_1 al poste p_3 es como sigue:

i) Se mueven $n - 1$ discos de p_1 a p_2, dejando sólo al disco n, el más grande, en p_1.
ii) Se mueve $n - 1$ el disco n de p_1 a p_3.
iii) Se mueven $n - 1$ discos de p_2 a p_2, para que descansen sobre el disco n.

o, expresando los pasos anteriores en forma de algoritmo:

Algoritmo 2 Torres de Hanoi

```
procedure HANOI(n, o, d, a)        ▷ Donde n es el número de discos,
                                     o es el poste origen,
                                     d es el poste destino y
                                     a es el poste auxiliar.
    if discos > 0 then
        HANOI(n − 1, o, a, d)
        Mueve n de o a d
        HANOI(n − 1, a, d, s)
    end if
end procedure
```

Esta solución es del orden $O(2^n)$.

4. Desarrollo de la práctica

Lea cuidadosamente los problemas descritos a continuación y desarrolle los programas propuestos.

4.1. Actividad 1

Con base en el Algoritmo 1, complete el programa esqueleto del Apéndice A.1 e implemente la función de búsqueda binaria `binsearch` de forma recursiva. Pruebe su programa con los siguientes conjuntos de datos:

```
python3 binsearch.py 100
python3 binsearch.py 69
```

Complete la siguiente tabla [2 puntos]:

key	**Encontrado**	**Posición**	*h*
29			
55			
69			
100			
3239			
12962			
44151			
50608			
74989			
99994			

4.2. Actividad 2

El programa del Apéndice A.2 imprime la secuencia de pasos para resolver el rompecabezas de las torres de Hanoi con 3 discos que se quieren mover del poste izquierdo (1) al central (2). Ejecútelo con la línea:

```
python3 hanoi.py
```

Analice el programa del Apéndice A.2 y, de ser necesario, modifíquelo para que imprima la secuencia de pasos para resolver el rompecabezas con 5 discos que deben moverse del poste derecho (3) al central (2).
¿Qué modificaciones se realizaron al programa? Explique: [1 punto]

¿Cuántos pasos se requieren para mover los 5 discos? Explique: [1 punto]

¿Cuántos pasos se requerirán para mover 8 discos? Modifique el programa en caso necesario y explique: [1 punto]

[Opcional] Pruebe que la complejidad del algoritmo es de $O(2^n)$ con n el número de discos. [+5 puntos]

4.3. Actividad 3

El programa del Apéndice A.3 determina si las palabras pasadas como argumentos son palíndromos o no, es decir, si las palabras pueden leerse igual de derecha a izquierda que de izquierda a derecha. Ejecútelo con las líneas:

```
python3 palindrome.py noon
python3 palindrome.py auch
```

Modifique el programa del Apéndice A.3 de forma que la función `palindrome` sea recursiva y pruebe el programa con las siquientes palabras:

- anilina
- arenera
- erre
- ese
- oso
- rodador
- solos
- someteremos

¿Qué modificaciones realizó al programa para hacerlo recursivo? Explique [5 puntos]: _____

_____

_____

[Opcional] Modifique su programa de manera que acepte palabras que mezclan mayúsculas y minúsculas y aún así verifique palíndromos. [+1 puntos].
[Opcional] Modifique su programa de manera que acepte espacios y caracteres acentuados del español además de mayúsculas y minúsculas. [+2 puntos].

A. Código de ejemplo

A.1. Archivo `src/binsearch.py`

src/binsearch.py

```python
import sys
from random import sample, seed

h = 0
def binsearch(key, A, l, r):
  global h
  h+=1

  ################################################
  #
  # Student's code here!
  #
  ################################################
# end def

def main(argv):
  # Generate random array A
  seed(69)
  limit = int(argv[2]) if len(argv) > 2 else 10000
  A = sorted(sample(range(0, limit*10), limit))

  # Fetch key
  key = int(argv[1]) if len(argv) > 1 else 69

  print("Searching for {} in A (n={})".format(key, len(A)-1))
  result = binsearch(key, A, 0, len(A))
  if result != -1:
    print("{} found at position {} ({})".format(key, result, A[result-2:result+3]))
  else:
    print("{} not found".format(key))
  print("h =", h)

if __name__ == "__main__":
  main(sys.argv)
```

A.2. Archivo `src/hanoi.py`

src/hanoi.py

```python
import sys

def hanoi(disks, src=1, dest=2, aux=3):
  if disks > 0:
    # Mueve n-1 discos del origen (src) al auxiliar
    # usando destino como poste intermedio
    hanoi(disks-1, src, aux, dest)

    # Mueve un disco del origen al destino
    print("Mueve disco D{} de P{} a P{}".format(disks, src, dest))

    # Mueve n-1 discos del auxiliar al destino
    # usando origen (src) como poste intermedio
    hanoi(disks-1, aux, dest, src)

def main(argv):
  n = int(argv[1]) if len(argv) > 1 else 3
  hanoi(n)

if __name__ == "__main__":
  main(sys.argv)
```

A.3. Archivo `src/palindrome.py`

src/palindrome.py

```python
import sys

def palindrome(s):
  i = 0
  j = len(s)-1
  while i < j:
    if s[i] != s[j]:
      return False
    i+=1
    j-=1
  return True
# end def


def main(argv):
  for s in argv[1:]:
    print("'{}' {} palíndromo.".format(
      s,
      "es" if palindrome(s) else "no es"
    ))
# end def

if __name__ == "__main__":
  main(sys.argv)
```

B. Reporte Escrito

El reporte de la práctica deberá ser entregada en un archivo en formato `PDF` siguiendo las siguientes especificaciones:

- La primera página del documento deberá ser la carátula oficial para prácticas de laboratorio disponible en `lcp02.fi-b.unam.mx/`
- El nombre del documento `PDF` deberá ser `nn-XXXX-L12.pdf`, donde:
  - `nn` es el número de lista del alumno a dos dígitos forzosos (ej. 01, 02, etc.).
  - `XXXX` corresponden a las dos primeras letras del apellido paterno seguidas de la primera letra del apellido materno y la primera letra del nombre, en mayúsculas y evitando cacofonías; es decir, los cuatro primeros caracteres de su RFC o CURP.
- El reporte consiste en un documento de redacción propia donde el alumno explica de forma concisa y a detalle las actividades realizadas en la práctica, y reportando los resultados obtenidos.
- La longitud del reporte no deberá exceder las 3 páginas, sin contar la carátula.
- El reporte deberá seguir todos los lineamientos para documentos escritos establecidos al inicio del curso.
- Todas las referencias deberán estar debidamente citadas.

IMPORTANTE: No se aceptan archivos en otros formatos ni con nombres distintos a los especificados.