Práctica 11: Estrategias para la construcción de algoritmos

Estructuras de Datos y Algoritmos I

Autor: José Mauricio Matamoros de Maria y Campos

1. Objetivo

El alumno aprenderá a identificar las diferentes estrategias para la construcción de algoritmos (fuerza bruta, codiciosos, programación dinámica y divide y vencerás) y sus paradigmas (*top-down* y *bottom-up*).

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

El diseño de algoritmos para resolver problemas de cómputo no es una tarea trivial. Cuando se diseña un algoritmo es necesario comprobar la correctez del mismo y calcular su complejidad.

- **Correctez:** Se dice que un algoritmo es correcto cuando éste es capaz de funcionar con cualquier conjunto de datos (incluyendo condiciones de frontera) y se puede demostrar matemáticamente que el algoritmo siempre terminará el cómputo en un tiempo finito. Además, un algoritmo tendrá que devolver siempre el mismo resultado cuando se le proporcionan dos conjuntos de datos idénticos.
- **Compejidad:** El cálculo de la complejidad de un algoritmo corresponde a la estimación matemática del tiempo de ejecución (o memoria requerida) del algoritmo en función del tamaño del conjunto de datos suministrado, aproximada mediante el cálculo de una familia de funciones asintóticas que lo contengan.

Al momento de diseñar algoritmos es conveniente basarse en alguna de las estrategias conocidas que simplifican el modelado de los algoritmos y hacen más sencillo su análisis. Estas estrategias se engloban dentro de dos grandes paradigmas: el analítico o *top-down* y el sintético o *bottom-up* que categorizan a los algoritmos según su aproximación y grado de abstracción.

Paradigma *top-down*

El paradigma analítico o *de arriba hacia abajo (top-down)* modela los problemas como estructuras complejos de alto grado de abstracción como sujetos de análisis. Así, los problemas se descomponen en sub-problemas más simples de manera sucesiva hasta que se obtiene un conjunto de problemas simples de solución casi trivial. Posteriormente estas soluciones simples se integran para obtener la solución general. Se llaman *top-down* porque van de arriba (un

alto grado de complejidad) hacia abajo (un grado ínfmo de complejidad) es decir, se parte de lo abstracto y se llega a lo simple.

Un ejemplo de este paradigma es el modelado de la función factorial $f(n) = n! \mid n \in \mathbb{N}$ expresada de forma recursiva como:

$$n! = \begin{cases} 0 & n \leq 1 \\ n \cdot (n-1)! & n > 1 \end{cases}$$

Nótese que en este caso la solución con $n \leq 1$ es trivial, mientras que los casos no triviales se pueden descomponer como el producto del número por el factorial de su predecesor. Así, en cada paso el problema se simplifica hasta llegar a una solución trivial conocida.

Paradigma *bottom-up*

El paradigma sintético o *de abaho hacia arriba (bottom-up)* sintetiza datos crudos agrupándolos en entidades con un grado de abstracción cada vez más compleja hasta obtener una solución al problema modelado. Se llaman *bottom-up* porque van de abajo (un grado ínfmo de complejidad) hacia arriba (un alto grado de complejidad) es decir, se parte de lo simple para construir algo abstracto.

Como los datos crudos suelen ser valores numéricos arbitrarios, es común el uso de métodos estadísticos y de técnicas de reconocimiento de patrones para su análisis a fin de dotarlos de sentido.

Un ejemplo de este paradigma es el modelado de la función factorial $f(n) = n! \mid n \in \mathbb{N}$ expresada como una serie de productos:

$$n! = \begin{cases} 0 & n \leq 1 \\ \prod_{i=1}^{n} i = 1 \times 2 \times \cdots \times n & n > 1 \end{cases}$$

Nótese que en este caso la solución se construye como una serie de productos hasta llegar al valor deseado. Así, en cada paso el problema acumula las soluciones conocidas con la nueva información hasta llegar a la respuesta.

3.1. Estrategia: Búsqueda exhaustiva

La solución de problemas mediante la estrategia de búsqueda exhaustiva consiste en generar todo el espacio solución del problema a partir de sus condiciones iniciales y buscar en éste por la solución óptima. En general, la búsqueda es ciega y ordenada.

Por ejemplo, considere que se desea generar un algoritmo que encuentre contraseñas. Sabemos que las contraseñas están formadas a partir de un alfabeto base L y tienen una determinada longitud k, permitiendo que se utilize cualquier combinación de los n símbolos s_i en el alfabeto $L = \{s_1, s_2, \ldots, s_n\}$ en grupos de tamaño k, incluyendo repeticiones. A este problema común se le conoce como *Permutaciones* y se calcula fácilmente con la fórmula:

$$_nP_k = n^k$$

Ahora bien, supóngase que se utiliza como alfabeto el de las letras minúsculas del idioma inglés (25) y el número de caracteres de la contraseña es de 3. Esto daría lugar a $25^3 = 15625$ combinaciones. Ahora bien, supóngase que se cuenta con una computadora que puede probar 1×10^9 combinaciones por segundo. Como el número de contraseñas a generar es pequeño, la computadora acabaría relativamente rápido, pero si se incrementa el número de caracteres a, digamos 8, el número de contraseñas a generar sería de $25^8 = 152587890625$, lo que a esta computadora le llevaría 152 segundos.

Aumentemos ahora el alfabeto agregando mayúsculas y números. De inmediato tenemos 60 símbolos con los que probar, por lo que una contraseña de 8 caracteres da lugar a $60^8 = 167961600000000$ combinaciones que requerirían 167961 segundos (aproximadamente 48 horas) para generarse. Si ahora cambiamos de longitud fija a variable el número de caracteres, sería equivalente a añadir un símbolo más (fin de cadena), dando lugar a $61^8 = 191707312997281$ combinaciones que de aproximadamente 53 para ser generadas.

Si se aumentara la longitud máxima de la contraseña hasta 16 caracteres y el alfabeto para incluir símbolos (ej. `!?-_.,;:[]`), vocales estresadas (ej. `åêíõüø`) y otros caracteres especiales de los alfabetos occidentales (ej. `ñßčçşł`) tanto en sus versiones mayúsculas como minúsculas, se tendría un alfabeto de 150 símbolos o más. El

número de cadenas posibles es de $150^{16} > 6.56 \times 10^{34}$ posibles combinaciones en el espacio solución, requiriendo un tiempo de cómputo no menor a 2.082×10^{18} años; aproximadamente 151 millones de veces la edad del universo.[1]

Fuerza bruta

Otro nombre que recibe la búsqueda completa en el espacio-solución es el de *Fuerza Bruta*, pues se prueban todas las combinaciones posibles. No obstante, algunos autores marcan una sutil diferencia que radica en que, mientras que búsqueda exhaustiva es ordenada, cuando se realiza una búsqueda por fuerza bruta se suele utilizar primero un diccionario o tabla con las soluciones más probables, o bien una generación aleatoria, por lo que el principio de órden se flexibiliza en favor de eurísticas que podrían acortar el tiempo de búsqueda con un poco de suerte, por ejemplo probando primero palabras como `password` o `P4$$w0Rd` al tratar de adivinar una contraseña.

3.2. Estrategia: Divide y vencerás

En la estrategia *divide-y-vencerás* los problemas se resuelven de manera recursiva aplicando tres pasos básicos en cada uno de los niveles de la recursión:

- **Dividir:** Se divide el problema en varios subproblemas que no son sino instancias más pequeñas del mismo problema.
- **Conquistar:** Se vence o derrota a cada subproblema de forma recursiva si el tamaño del problema es demasiado grande pero, cuando es suficientemente pequeño, se le resuelve de forma directa.
- **Combinar:** Se combinan las soluciones de los subproblemas en una solución para el problema original.

Cuando los subproblemas son lo suficientemente grandes como para resolverse recursivamente se le llama «caso recursivo» y una vez que los subproblemas son lo suficientemente pequeños como para hacer innecesaria la recursión se dice que la recursividad *toca fondo* y que se ha llegado al caso base. A veces, además de los subproblemas, que son instancias más pequeñas del mismo problema, se tiene que resolver subproblemas que no son exactamente iguales al problema original. La resolución de estos subproblemas se considera parte del paso de combinación.

QuickSort

Un ejemplo socorrido de un algoritmo que utiliza la estrategia *divide-y-vencerás* es el algoritmo *QuickSort* o de ordenamiento rápido que se presenta a continuación:

Algoritmo 1 Quicksort

```
1: procedure QUICKSORT(A, p, r)                ▷ A: conjunto de elementos a ordenar
2:     if p < r then
3:         q ← PARTITION(A, p, r)
4:         QUICKSORT(A, p, q − 1)              ▷ Ordena todos los elementos menores o iguales q
5:         QUICKSORT(A, q + 1, r)              ▷ Ordena todos los elementos mayores o iguales a q
6:     end if
7: end procedure
```

El algoritmo funciona como sigue:

i) **Dividir:** Particiona (reacomoda) el arerglo $A[p \dots r]$ en dos sub-arreglos (que pueden ser vacíos) $A[p \dots q-1]$ y $A[q+1 \dots r]$ tales que $\forall x \in A[p \dots q-1], x \leq A[q]$ y $\forall y \in A[q+1 \dots r], y \geq A[q]$. Calcular el elemento q o *pivote* es parte del proceso de partición.

ii) **Conquistar:** Ordena los dos sub-arreglos $A[p \dots q-1]$ y $A[q+1 \dots r]$ con llamadas recursivas a *QuickSort*.

iii) **Combinar** Como los sub-arreglos ya están ordenados, no es necesaripo hacer nada. El arreglo completo $A[p \dots r]$ estará ordenado al final de la ejecución.

El *quid* del algoritmo radica en el procedimiento PARTITION que elige un pivote q y reacomoda los elementos de A.

[1] Se estima que la edad del universo es de 1,377 millones de años.

Algoritmo 2 Partition Procedure

```
1: procedure PARTITION(A, p, q, r)                ▷ A: conjunto de elementos a reordenar
2:     x ← A[r]
3:     i ← p − 1
4:     for j ← p to r − 1 do
5:         if A[j] ≤ x then
6:             i ← i + 1
7:             swap A[i] ⟷ A[j]
8:         end if
9:     end for
10:    swap A[i + 1] ⟷ A[r]
11:    return i + 1
12: end procedure
```

3.3. Estrategia: Programación dinámica

La programación dinámica es similar a la estrategia *divide y vencerás* pues resuelve problemas combinando las soluciones de sus subproblemas. Nótese que, en este contexto, *programar* se refiere a un método tabular, no a escribir código de computadora. Sin embargo, a diferencia de *divide y vencerás* la programación dinámica se aplica cuando los subproblemas se superponen, es decir, cuando los subproblemas comparten subproblemas. Es decir que, desde el punto de vista de la programación dinámica, un algoritmo *divide y vencerás* hace más trabajo del necesario, resolviendo repetidamente los subproblemas comunes. Un algoritmo de programación dinámica resuelve cada subproblema solo una vez y luego guarda su respuesta en una tabla, evitando así el trabajo de volver a calcular la respuesta cada vez que resuelve cada subproblema.

Normalmente se aplica la estrategia de programación dinámica a los **problemas de optimización**. Estos problemas pueden tener muchas soluciones posibles, cada solución tiene un valor y se desea encontrar una solución con el valor óptimo (mínimo o máximo). A cada solución local se le suele llamar siplemente *una* solución óptima en contraste con ***la*** solución óptima, ya que puede haber muchas soluciones que generen valores óptimos pero sólo una entre ellas será ***la solución óptima***.

Al desarrollar un algoritmo de programación dinámica, se sigue una secuencia de cuatro pasos:

- Se caracteriza la estructura de una solución óptima.
- Se define de forma recursiva el valor de una solución óptima.
- Se calcula el valor de una solución óptima, normalmente de forma sintética o ascendente (bottom-up).
- Se construye una solución óptima a partir de información calculada.

Los pasos 1 a 3 forman la base de una solución de programación dinámica a un problema. Si solo se necesita el valor de una solución óptima y no la solución en sí, se puede omitir el paso 4. Cuando se lleva a cabo el paso 4 es a menudo necesario almacenar información adicional del paso 3 para poder construir fácilmente una solución óptima.

3.4. Estrategia: Algoritmos codiciosos

Esta estrtegia se aplica a algoritmos que resuelven problemas de optimización y que se dividen en una secuencia de pasos donde en cada paso existe conjunto de opciones. En este tipo de algoritmos, a menudo encontrar la solución óptima en todos los casos no importa (una suficientemente buena bastará), y utilizar programación dinámica para determinar la mejor opción es excesivo (requiere mucho tiempo), pues otros algoritmos más simples, o incluso heurísticas son más que suficiente.

Un *algoritmo codicioso* siempre tomará la decisión que se ve mejor en ese momento. Es decir, el algoritmo toma la opción óptima a nivel local con la esperanza de que esta elección conduzca a una solución óptima a nivel global.

Los algoritmos codiciosos no siempre producen soluciones óptimas, aunque para muchos problemas lo hacen. Ejemplos de algoritmos codiciosos son: i) diseño de códigos de compresión de datos (Huffman) ii) programación de tareas de tiempo unitario con fechas límite y penalizaciones iii) árbol de expansión mínimo y iv) el algoritmo de Dijkstra para la ruta más corta.

4. Desarrollo de la práctica

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

4.1. Actividad 1

Los programas de los Apéndices A.1 y A.2 calculan el factorial de un número entero positivo n. Ejecútelos con las líneas:

```
python3 fact1.py 10
python3 fact2.py 10
```

Compare los programas de los Apéndices A.1 y A.2 y responde las siguientes preguntas:

¿Qué paradigma utiliza el programa del Apéndice A.1? Explique [1 punto]: _____

¿Qué paradigma utiliza el programa del Apéndice A.2? Explique [1 punto]: _____

4.2. Actividad 2

El programa del Apéndice A.3 mide e imprime el tiempo que tarda en ejecutarse la función `foo()`. Ejecútelo con la línea:

```
python3 time.py
```

Combine el programa del Apéndice A.3 con los programas de los Apéndices A.1 y A.2 para poder medir el tiempo que tarda cada uno de los programas en calcular el factorial de un número entero. Con la información obtenida llene la siguiente tabla.

n	Prog1.py	Prog2.py
10		
100		
500		
1000		
10000		

¿Existe alguna diferencia en la ejecución de los programas? ¿Alguno es más rápido? ¿Es posible llenar la tabla? Explique: [1 punto] _____

4.3. Actividad 3

El programa del Apéndice A.4 recibe una cantidad numérica como argumento que representa el vuelto o cambio a entregar, y devuelve el número de monedas que deben entregarse. Ejecútelo con la línea:

```
python3 change.py 23.50
```

Este programa (Apéndice A.4) tiene una limitación muy grave: asume que tiene disponible cambio infinito, por lo que siempre da un número mínimo de monedas que podrían no estar disponibles.

Modifique el programa del Apéndice A.4 para que entregue la menor cantidad de monedas posibles usando sólo las monedas disponibles. Para tal fin, reemplace la línea 10 (variable `coins`) con:

```
coins = {
  0.1 : 3,
  0.2 : 3,
  0.5 : 2,
    1 : 0,
    2 : 3,
    5 : 0,
   10 : 3,
   20 : 3,
   50 : 3,
  100 : 3,
}
```

Pruebe su programa con:

```
python3 change.py  23.50
python3 change.py 171.30
python3 change.py  19.90
```

¿Qué paradigma y qué estrategia utiliza el programa? Explique [1 puntos]: _____

¿Qué modificaciones se realizaron al programa? Explique [2 puntos]: _____

4.4. Actividad 4

Los programas de los Apéndices A.4 y A.5 ordenan un arreglo de números generados aleatoriamente usando dos algoritmos: InsertionSort y QuickSort. Ejecútelos con las líneas:

```
python3 isort1.py 100
python3 qsort2.py 100
```

Posteriormente complete la siguiente tabla anotando los tiempos de ordenamiento:

n	isort.py	qsort.py
10		
100		
1000		
10000		
100000		
1000000		

¿Qué paradigma y qué estrategia utiliza QuickSort? Explique [1 punto]: _____

Con base en la información de la tabla, genere una gráfica de dispersión $n, t_i(n), t_q(n)$ donde compare los tiempos de ejecución de los algoritmos *QuickSort* e *Insertion Sort* en función del tamaño del arreglo (n). Analice los resultados y explique las diferencias [3 puntos]: _____

[2 puntos extra:] Utilice matplotlib para generar las gráficas.
[3 puntos extra:] Una los programas de los Apéndices A.4 y A.5 y automatice la generación de las gráficas con *matplotlib* y *numpy*, iterando en el intervalo $n \in [0, 10000]$ con un incremento $k = 10$ (`n in range(10, 10001, 10)`).

A. Código de ejemplo

A.1. Archivo `src/fact1.py`

src/fact1.py

```python
import sys

def fact(n):
  f = 1
  for i in range(2, n+1):
    f*= i
  return f

def main(argv):
  n = int(argv[1])
  print(n, "! = ", fact(n), sep="")


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

A.2. Archivo `src/fact2.py`

src/fact2.py

```python
import sys

def fact(n):
  return 1 if n <= 1 else n * fact(n-1)

def main(argv):
  n = int(argv[1])
  print(n, "! = ", fact(n), sep="")


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

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

src/time.py

```python
from time import perf_counter, sleep

def foo():
  sleep(1)

def main():
  start = perf_counter()
  foo()
  elapsed = perf_counter() - start
  print("Foo took {:.2f}ms".format(elapsed*1000))


if __name__ == '__main__':
  main()
```

A.4. Archivo `src/change.py`

src/change.py

```
import sys
coins = [0.1, 0.2, 0.5, 1, 2, 5, 10]

def change(amount):
  i = len(coins) -1
  my_change = [[coin, 0] for coin in coins]
  while(amount > 0 and i >= 0):
    if amount >= coins[i]:
      amount-= coins[i]
      my_change[i][1]+=1
    else:
      i-= 1
  return my_change
# end def


def main(argv):
  amount = float(argv[1])
  print("Cambio de", amount, "en:")
  for (coin, num) in sorted(change(amount), reverse=True):
    if not num:
      continue
    print(num,
      "moneda" if num == 1 else "monedas",
      "de",
      coin if coin >= 1 else coin * 10,
      "pesos" if coin >= 1 else "centavos")

# end def


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

A.5. Archivo `src/isort.py`

src/isort.py

```python
import sys
from random import randint, seed
from time import perf_counter

def swap(A, i, j):
  tmp = A[i]
  A[i] = A[j]
  A[j] = tmp
# end def


def insertionsort(A):
  for i in range(1, len(A)):
    key = A[i]
    j = i-1
    while j >= 0 and A[j] > key:
      A[j+1] = A[j]
      j-=1
    A[j+1] = key
# end def


def main(argv):
  seed(69)
  n = int(argv[1]) or 100
  A = [randint(-n, n+1) for i in range(n)]
  # print(A)
  start = perf_counter()
  insertionsort(A)
  elapsed = perf_counter() - start
  # print(A)
  print("Insertion sort time {:.2f}ms".format(elapsed*1000))


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

A.6. Archivo `src/qsort.py`

src/qsort.py

```python
import sys
from random import randint, seed
from time import perf_counter

def swap(A, i, j):
  tmp = A[i]
  A[i] = A[j]
  A[j] = tmp
# end def


def quicksort(A, p, r):
  if p < r:
    q = partition(A, p, r)
    quicksort(A, p, q-1)
    quicksort(A, q+1, r)
# end def


def partition(A, p, r):
  x = A[r]
  i = p-1
  for j in range(p, r):
    if A[j] <= x:
      i+= 1
      swap(A, i, j)
  swap(A, i+1, r)
  return i+1
# end def


def main(argv):
  seed(69)
  n = int(argv[1]) or 100
  A = [randint(-n, n+1) for i in range(n)]
  # print(A)
  start = perf_counter()
  quicksort(A, 0, len(A)-1)
  elapsed = perf_counter() - start
  # print(A)
  print("Quicksort time {:.2f}ms".format(elapsed*1000))


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-L11.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.