Técnicas Algorítmicas – Guía de Ejercicios

Master in Management + Analytics, Escuela de Negocios, UTDT

Segundo semestre 2020

1. Recursión

Ejercicio 1.1. Programar las siguientes funciones recursivas en Python:

(a) `pot_dos(n)`, que dado un número $n \in \mathbb{N}$ calcule 2^n. Ayuda: $2^0 = 1$.

(b) `pot_a(a,n)`, que dados $a, n \in \mathbb{N}$ calcule a^n.

Ejercicio 1.2. La sucesión de Fibonacci es una secuencia de números en la cual los dos primeros términos son 1 y los siguientes se definen como la suma de los dos anteriores. A continuación se observan los primeros diez términos de la sucesión:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

Programar una función *recursiva* en Python `fibonacci(n)` que dado un número $n \in \mathbb{N}$, devuelva el n-ésimo término de la sucesión de Fibonacci.

Ejercicio 1.3. Programar una función *recursiva* en Python `es_par(n)` que determine si un número $n \in \mathbb{N}$ es par (*i.e.*, que devuelva `True` si es par y `False` en caso contrario).

Ejercicio 1.4. Programar una función *recursiva* en Python `productoria(l)` que dada una lista no vacía de enteros l devuelva el resultado de multiplicar todos los números de l.

Ejercicio 1.5. Programar una función *recursiva* en Python `cantidad_ocurrencias(n,l)` que dada una lista de enteros l y un número n, devuelva la cantidad de ocurrencias de n en l.

Ejercicio 1.6. Estimar la complejidad temporal de los algoritmos de los ejercicios anteriores. En los ejercicios 1.4 y 1.5, si se usó la operación *slice* de listas, `l[i:j]`, tener en cuenta que la misma tiene $O(j-i)$; reescribir esas dos funciones de manera que su complejidad sea lineal en la longitud de la lista de entrada.

2. Divide & Conquer

Ejercicio 2.1. Programar una función *divide and conquer* en Python `pot_a(a,n)` que dados $a, n \in \mathbb{N}$ calcule a^n. Ayuda: $a^n = a^{\lceil \frac{n}{2} \rceil} \times a^{\lfloor \frac{n}{2} \rfloor}$. Sugerencia: investigar las funciones `ceil` y `floor` del módulo `math` de Python.

Ejercicio 2.2. Programar una función *divide and conquer* en Python `max_pos(l)` que dada una lista no vacía de enteros l devuelva la posición del elemento más grande. En caso de que el valor máximo esté en más de una posición, devolver la primera de ellas.

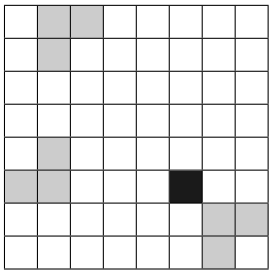
Ejercicio 2.3. Sea A una lista de números enteros cuya longitud es potencia de 2 (es decir: 2, 4, 8, 16, 32, etc.). Decimos que A es *"más a la izquierda"* si:

(I) la suma de los elementos de la mitad izquierda de A es mayor a la suma de los elementos de la mitad derecha de A;

(II) cada una de las mitades de A es, a su vez, "más a la izquierda".

Por ejemplo, [8, 6, 7, 4, 5, 1, 3, 2] es "más a la izquierda", pero [8, 4, 7, 6, 5, 1, 3, 2] no. Programar una función *divide and conquer* en Python `es_mas_a_la_izquierda(l)` que determine si una lista l es "más a la izquierda".

Ejercicio 2.4. *(Opcional)*
Un *L-rompecabezas* es un juego en el que hay un tablero de $2^n \times 2^n$ casilleros para algún $n \in \mathbb{N}$ (*i.e.*, el tablero puede ser de 2×2, 4×4, 8×8, 16×16, etc.) y *L-fichas* en forma de L que pueden cubrir tres celdas adyacentes. Además, el tablero contiene una posición distinguida llamada *agujero negro*. A modo de ejemplo, el siguiente dibujo representa un tablero de 8×8 con tres *L-fichas*:



El objetivo del juego consiste en, dado un tablero vacío con exactamente un *agujero negro*, usar *L-fichas* para cubrir todas las posiciones del tablero, sin superponer *L-fichas* y sin cubrir el *agujero negro*. Se pide diseñar una estrategia *divide and conquer* para resolverlo. (Pensar el algoritmo de manera abstracta; no hace falta implementarlo en Python.)

3. Programación Dinámica

Ejercicio 3.1. Dada una lista de enteros l, queremos determinar en cuántas posiciones el valor coincide con la suma de los valores de las posiciones anteriores. Tenemos el siguiente algoritmo que resuelve el problema en tiempo cuadrático respecto de la longitud de l:

```
1  def cant_pos_con_suma_ant(l):
2    res, i = 0, 0
3    while i < len(l):
4       suma_ant, j = 0, 0
5       while j < i:
6          suma_ant = suma_ant + l[j]
7          j = j + 1
8       if suma_ant == l[i]:
9          res = res + 1
10      i = i + 1
11   return res
```

Por ejemplo, `cant_pos_con_suma_ant([1,1,2,3,7,0,14])` devuelve 4. Se pide escribir un algoritmo de complejidad lineal, empleando la técnica de programación dinámica.

Ejercicio 3.2. Considérese el problema de, dado un número $n \in \mathbb{N}$, determinar de cuántas formas puede escribirse como suma de los números 1, 3 y 4. Por ejemplo, para $n = 5$, la respuesta es 6:

- $5 = 1 + 1 + 1 + 1 + 1$
- $5 = 1 + 4$
- $5 = 4 + 1$
- $5 = 1 + 1 + 3$
- $5 = 1 + 3 + 1$
- $5 = 3 + 1 + 1$

Se cuenta con la siguiente función en Python `cant_sumas_distintas_exp(n)` que resuelve el problema, pero en tiempo exponencial:

```
def cant_sumas_distintas_exp(n):
    if n == 1:
        rv = 1      # hay 1 forma de escribir 1: (1)
    elif n == 2:
        rv = 1      # hay 1 forma de escribir 2: (1+1)
    elif n == 3:
        rv = 2      # hay 2 formas de escribir 3: (1+1+1),(3)
    elif n == 4:
        rv = 4      # hay 4 formas de escribir 4: (1+1+1+1),(1+3),(3+1),(4)
    else:
        rv = cant_sumas_distintas_exp(n-1) + \
             cant_sumas_distintas_exp(n-3) + \
             cant_sumas_distintas_exp(n-4)
    return rv
```

(a) Explicar por qué este algoritmo resuelve el problema y por qué es exponencial.

(b) Escribir una función en Python `cant_sumas_distintas_lineal(n)` $\in O(n)$ que también resuelva el problema.

(c) Comparar los tiempos de ejecución de ambas funciones (la dada y la propuesta) tomando $n = 30$, $n = 35$, $n = 40$ y $n = 45$. ¿A qué se deben las diferencias observadas?

Ejercicio 3.3. Laura y Lucas son dos amigos que suelen jugar al *bolicolores*. Se trata de un juego en el que, alternadamente, van sacando bolitas de una bolsa, de a una por vez. Cuando una bolita sale de la bolsa, ya no se la vuelve a poner en la misma. Inicialmente, en la bolsa hay $r \geq 0$ bolitas rojas y $v \geq 0$ bolitas verdes. El primero en sacar una bolita roja gana la partida. Además, siempre es Laura quien retira la primera bolita y, si luego de sacar todas no apareció ninguna roja, gana Lucas.

El problema consiste en, dados r y v, calcular la probabilidad (un número entre 0 y 1) de que gane Laura. Se muestra a continuación una solución $\in O((r + v)^2)$ para resolver el problema:

```
# Devuelve la probabilidad de que Laura gane el juego; es decir,
# la probabilidad de que la primera bolita roja salga en el primer
# turno, o en el tercero, o en el quinto, etc.
def prob_laura(r, v):
    ret = 0
    turno = 1
    while turno <= r+v:
        ret = ret + prob_primera_roja_en_turno(r, v, turno)
        turno = turno + 2
```

```
10      return ret
11
12  # Dadas r bolitas rojas y v bolitas verdes, devuelve la probabilidad
13  # de que la primera bolita roja salga en el turno indicado; es decir, la
14  # probabilidad de que primero salgan (turno-1) verdes y luego una roja.
15  def prob_primera_roja_en_turno(r, v, turno):
16      ret = 1
17      i = 1
18      while i < turno:
19          ret = ret * prob_verde(r, v)
20          v = v - 1
21          i = i + 1
22      ret = ret * prob_roja(r, v)
23      return ret
24
25  # Dadas r bolitas rojas y v bolitas verdes,
26  # devuelve la probabilidad de sacar una bolita roja.
27  def prob_roja(r, v):
28      if r == 0:
29          ret = 0
30      else:
31          ret = r/(r+v)
32      return ret
33
34  # Dadas r bolitas rojas y v bolitas verdes,
35  # devuelve la probabilidad de sacar una bolita verde.
36  def prob_verde(r, v):
37      if v == 0:
38          ret = 0
39      else:
40          ret = v/(r+v)
41      return ret
```

(a) Escribir una función en Python `prob_laura_iter(n)` $\in O(r+v)$ iterativa que también resuelva el problema.

(b) (Opcional) Escribir una función en Python `prob_laura_rec(n)` $\in O(r+v)$ recursiva que también resuelva el problema.

4. Backtracking

Ejercicio 4.1. Dado un conjunto C, se dice que C' es un *subconjunto* de C si y solamente si todo elemento que está en C' también está en C. Además, el *conjunto de partes* $\wp(C)$ es el conjunto cuyos elementos son los subconjuntos de C. Por ejemplo, el conjunto de partes de $C = \{1, 2\}$ es $\big\{\{\}, \{1\}, \{2\}, \{1, 2\}\big\}$.

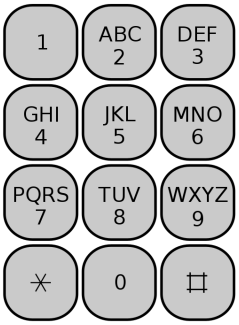
Programar una función *backtracking* en Python `conjunto_de_partes(c)` que dado un conjunto C devuelva $\wp(C)$. Sugerencia: investigar los tipos `set` y `frozenset` de Python. A continuación se muestran posibles entradas y, en cada caso, el resultado esperado:[1]

- $\{0\} \longmapsto \big\{\{\}, \{0\}\big\}$

[1] El orden en el que se imprimen los elementos de un conjunto es irrelevante.

- $\{1,2\} \longmapsto \big\{\{\},\{1\},\{2\},\{1,2\}\big\}$
- $\{\} \longmapsto \big\{\{\}\big\}$

Ejercicio 4.2. En la actualidad muchos comercios expresan sus números de teléfono como una palabra que puede formarse con las letras que figuran en los teclados telefónicos. Por ejemplo, una famosa casa de empanadas cuyo número es `3672-6232` difunde su teléfono como `EMPANADA` en lugar del número de ocho dígitos difícil de recordar. La correspondencia entre dígitos y letras de un teclado telefónico puede observarse en la siguiente figura:



Nótese que para un mismo número existen muchas palabras posibles (la mayoría de ellas sin sentido).

(a) Escribir una función en Python `obtener_letras(d)` que dado un dígito d devuelva el conjunto de letras que se asocian a d en un teclado telefónico. A continuación se encuentran algunas invocaciones y el resultado esperado:

- `obtenerLetras(2)` → `{'A','B','C'}`
- `obtenerLetras(7)` → `{'P','Q','R','S'}`
- `obtenerLetras(0)` → `{ }`

(b) Escribir una función en Python `palabras_telefonicas(n, digitos)` que dados un número n y un conjunto de dígitos c, escriba en un archivo `palabras_telefonicas.out` todas las cadenas de caracteres de longitud n que pueden formarse con un teclado telefónico usando solo los dígitos que se encuentran en c. Sugerencia: utilizar la función definida en el punto (a).

Ejercicio 4.3. Una empresa utiliza un camión para mover mercadería entre sus dos depósitos. Antes de cada viaje, se debe decidir qué paquetes de mercadería cargar en el camión, de modo de aprovechar el viaje al máximo. El camión tiene una capacidad máxima de m metros cúbicos. Cada paquete a transportar tiene un volumen (en metros cúbicos) y un puntaje numérico entre 1 y 100 (calculado en base a criterios internos de la empresa, tales como la urgencia del envío).

El problema consiste en, dada una lista de paquetes que deben ser transportados, elegir una combinación de paquetes que quepa en el camión y maximice la suma de los puntajes de los paquetes transportados.[2] Para resolver el problema se cuenta con una implementación de los tipos de datos `Paquete` y `Camion` (ver el archivo adjunto `camion.py`). Se pide:

(a) Escribir en Python una función `cargar_camion(camion, inventario)`, que dados un camión y una lista de paquetes, devuelva un nuevo camión cargado con la mejor combinación de paquetes: es decir, la combinación de paquetes con puntaje máximo que cabe en el camión. Esta función debe realizar una búsqueda exhaustiva con la técnica de *backtracking*.

[2] Este problema es una variante del clásico *problema de la mochila, o "knapsack problem"*.

(b) Correr la función con inventarios de 10, 11, 12, 13, 14, 15 y 16 paquetes, agregando de a un paquete cada vez. Analizar y explicar a qué se debe la diferencia entre los tiempos necesarios para hallar la solución del problema para cada conjunto de paquetes.

Medir el tiempo de ejecución mediante:

```
import datetime
start = datetime.datetime.now()
print(cargar_camion(cam, inv))
print(datetime.datetime.now()-start)
```

(c) No es inusual que un restaurante que hace envíos a domicilio, en un determinado momento tenga 80 porciones de comida por repartir. ¿Cuánto tiempo demandaría encontrar la solución óptima para el servicio de reparto del local? ¿Sería admisible para un cliente esperar a que el programa termine de correr para recibir su pedido? ¿No moriría antes de inanición? ¿Qué sucedería, entonces, si se considera una empresa de logística que tiene que repartir 5000 cajas?

5. Heurísticas

Ejercicio 5.1. Considerar el problema del ejercicio 4.3 de carga de mercadería en un camión.

(a) Escribir una función `cargar_camion_g(camion, inventario)` que encuentre una solución aproximada (es decir, una combinación de paquetes que quepa en el camión pero que no necesariamente sea la de puntaje máximo) usando la heurística *greedy*.

(b) Escribir una función `cargar_camion_hc(camion, inventario)` que encuentre una solución aproximada usando una heurística *hill climbing*, tomando como punto de partida a una solución aproximada encontrada con el algoritmo *greedy* del punto anterior.

(c) Escribir una función `cargar_camion_hc2(camion, inventario, iter)` que repita una cantidad `iter` de veces la heurística *hill climbing*, utilizando cada vez una combinación de paquetes generada al azar como punto de partida.

(d) Comparar los tiempos de ejecución requeridos y la calidad de las soluciones encontradas en cada caso.