



5

Instituto Politécnico Nacional
Escuela Superior de Computo

Análisis de algoritmos

Sandra Díaz Santiago

Practica 7

García González Aarón Antonio

Noviembre 18, 2019



Índice

Ejercicio 1.....	3
Algoritmo por fuerza bruta.....	3
Implementación	3
Ejecución	6
Justificación	7
Algoritmo eficiente	7
Implementación	7
Ejecución	7
Justificación	7
Ejercicio 2.....	7
Algoritmo por fuerza bruta (Recursivo)	8
Implementación	8
Ejecución	8
Justificación	8
Algoritmo eficiente	9
Implementación	9
Ejecución	11
Justificación	11

Ejercicio 1

Dado un entero n , encontrar cuántas formas distintas hay de escribir n como la suma de 1, 3 y 4. Por ejemplo si $n = 5$, hay 6 formas distintas:

$$\begin{aligned} 5 &= 1 + 1 + 1 + 1 + 1 \\ &= 1 + 1 + 3 \\ &= 1 + 3 + 1 \\ &= 3 + 1 + 1 \\ &= 1 + 4 \\ &= 4 + 1 \end{aligned}$$

Diseña un algoritmo por fuerza bruta que resuelva el problema anterior y escribe el pseudocódigo y determina la complejidad de este. Posteriormente diseña un algoritmo de complejidad $O(n)$ para resolverlo. Explica mediante un ejemplo, por qué tu algoritmo funciona.

Algoritmo por fuerza bruta

Implementación

Primeramente, el “main” del programa:

```
# ----- Importacion de modulos a utilizar
from archivos_1 import *
from utilidades_1 import *

# ----- Declaracion de apuntador a archivo
name_file_INPUT = input("Teclee el nombre del archivo: ")

# ----- Ejecutable
n, nums = leerDatos(name_file_INPUT)
posibles = []

combinaciones = generaCombinaciones(n,nums) + llenaCombinacion(n,nums) + obtenerIguales(n,nums)
for i in combinaciones:
    i.sort()

combinaciones = set(tuple(x) for x in combinaciones)
combinaciones = [ list(x) for x in combinaciones ]
combinaciones_finales = generaCombinacionesFinales(combinaciones)

print("\nNúmero a calcular combinaciones: ", n)
print("Números a ocupar en combinaciones: ", nums)
print("Número de combinaciones: ", len(combinaciones_finales))
print("Posibles combinaciones ", combinaciones_finales, "\n")
```

A partir de un archivo de texto se leen los datos necesarios (Número a calcular combinaciones y los números que pueden ocupar las combinaciones)

```
p7 > ≡ input_1.txt
1      Nmero a calcular
2      5
3      Numeros candidatos
4      1 3 4
```

```
import itertools as it

# Funcion que retorna la suma de cada uno de los elementos de una list
# Recibe como parametro la lista
def obtenerSuma(sequencia):
    cont = 0
    for i in sequencia:
        cont += i

    return cont

# Funcion que rellena una lista con el numero minimo de otra lista
# Recibe como parametro el numero n, los numeros posibles, y las combinaciones obtenidas previamente
def llenaConMinimo(n,nums,combinaciones):
    for sequencia in combinaciones:
        while(obtenerSuma(sequencia) < n):
            sequencia.append(min(nums))

# Funcion que genera una lista de listas, cada una de estas listas contiene
# numeros iguales con las que se puede representar x
# Recibe como parametro el numero x y la lista de numeros posibles
def obtenerIguales(x,numeros):
    # posibles juntos
    result = []
    for i in numeros:
        cont = x
        aux = []
        while (cont/i >= 1):
            cont -= i
            aux.append(i)

        if len(aux) > 1:
            result.append(aux)
    llenaConMinimo(x,numeros,result)
    return result
```

Funcion que genera una lista con las posibles combinaciones entre el minimo y los demas numeros
Recibe como parametro el numero a calcular combinaciones y la lista de numeros disponibles

def generaCombinaciones(x,nums):

result = []

for i in nums:

aux = []

if i != min(nums):

for j in range(1,int(x/i)+1):

aux = []

for k in range(0,j):

aux.append(i)

result.append(aux)

llenaConMinimo(x,nums,result)

return result

Funcion que genera una lista con las posibles combionaciones entre nuemros que no contienen el minimo
Recibe como parametro l nuemro a calcular combinaciones y la lista de numeros disponibles

def llenaCombinacion(n,nums):

posibles = []

for i in range(2,len(nums)+1):

aux = list(it.permutations(nums,i))

cont = 0

for sec in aux:

cont = 0

for ind in sec:

cont += ind

if cont <= n:

posibles.append(list(sec))

for i in posibles:

i.sort()

posibles = set(tuple(x) **for** x in posibles)

posibles = [list(x) **for** x in posibles]

llenaConMinimo(n,nums,posibles)

return posibles

Funcion que genera las combinaciones totales que dan solucion al problema
Recibe como parametro todas las optenidas por la funcion "llenaCombinacion"
para posteriormente obtener permutaciones y discriminar todas las direntes

def generaCombinacionesFinales(candidatas):

result = []

for secuencia in candidatas:

aux = list(it.permutations(secuencia))

aux = set(tuple(x) **for** x in aux)

```
aux = [ list(x) for x in aux ]  
for sec in aux:  
    result.append(sec)  
  
return result
```

Ejecución

```
MacBook-Pro-de-Aaron:p7 aarongarcia$ python3 index_1.py
Teclee el nombre del archivo: input_1.txt

Número a calcular combinaciones: 5
Números a ocupar en combinaciones: [1, 3, 4]
Número de combinaciones: 6
Posibles combinaciones [[1, 1, 3], [1, 3, 1], [3, 1, 1], [1, 1, 1, 1, 1], [4, 1], [1, 4]]
```

```
MacBook-Pro-de-Aaron:p7 aarongarcia$ python3 index_1.py
Teclee el nombre del archivo: input_1.txt

Número a calcular combinaciones: 7
Números a ocupar en combinaciones: [1, 3, 4]
Número de combinaciones: 15
Posibles combinaciones [[1, 1, 1, 4], [1, 4, 1, 1], [4, 1, 1, 1], [1, 1, 4, 1], [1, 1, 1, 1, 1, 1, 1], [3, 4], [4, 3], [1, 3, 1, 1, 1], [1, 1, 1, 1, 3], [1, 1, 3, 1, 1], [1, 1, 1, 3, 1], [3, 1, 1, 1, 1], [3, 1, 3], [3, 3, 1], [1, 3, 3]]
```

```
MacBook-Pro-de-Aaron:p7 aarongarcia$ python3 index_1.py
Teclee el nombre del archivo: input_1.txt

Número a calcular combinaciones: 8
Números a ocupar en combinaciones: [1, 3, 4]
Número de combinaciones: 25
Posibles combinaciones [[4, 4], [1, 1, 1, 1, 1, 3], [1, 1, 3, 1, 1, 1], [3, 1, 1, 1, 1, 1], [1, 3, 1, 1, 1, 1], [1, 1, 1, 1, 3, 1], [1, 1, 1, 3, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1], [3, 1, 3, 1], [1, 1, 3, 3], [3, 1, 1, 3], [1, 3, 1, 3], [1, 3, 3, 1], [3, 3, 1, 1], [3, 1, 4], [4, 3, 1], [1, 3, 4], [1, 4, 3], [4, 1, 3], [3, 4, 1], [1, 4, 1, 1, 1], [4, 1, 1, 1, 1], [1, 1, 1, 4, 1], [1, 1, 4, 1, 1], [1, 1, 1, 1, 4]]
```

```
MacBook-Pro-de-Aaron:p7 aarongarcia$ python3 index_1.py
Teclee el nombre del archivo: input_1.txt

Número a calcular combinaciones: 9
Números a ocupar en combinaciones: [1, 3, 4]
Número de combinaciones: 40
Posibles combinaciones [[4, 1, 4], [1, 4, 4], [4, 4, 1], [1, 1, 1, 3, 3], [1, 1, 3, 3, 1], [3, 1, 1, 1, 3], [1, 3, 1, 3, 1], [3, 1, 3, 1, 1], [1, 3, 1, 1, 3], [3, 1, 1, 3, 1], [1, 1, 3, 1, 3], [1, 3, 3, 1, 1], [3, 3, 1, 1, 1], [1, 4, 3, 1], [1, 1, 4, 3], [4, 1, 3, 1], [1, 1, 3, 4], [3, 1, 1, 4], [1, 3, 1, 4], [3, 1, 4, 1], [1, 3, 4, 1], [3, 4, 1, 1], [4, 3, 1, 1], [4, 1, 1, 3], [1, 4, 1, 3], [3, 3, 3], [3, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 3], [1, 1, 1, 1, 1, 1, 1, 3], [1, 1, 1, 3, 1, 1, 1], [1, 3, 1, 1, 1, 1], [1, 1, 3, 1, 1, 1], [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 4, 1, 1], [4, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 4], [1, 4, 1, 1, 1, 1]]
```

[illegible]

Justificación

Dado que se usó la función permutaciones de Python, la complejidad del programa que es $O(n!)$, esto a que si recordamos la manera en que calculamos permutaciones pues es con operaciones factoriales, por lo que al aumentarse n , el algoritmo incrementa en orden factorial su tiempo de ejecución, es decir, para $n \geq 12$ se observa un tiempo muy grande de procesamiento.

Algoritmo eficiente

Implementación

```
result = []
result.append(0)
result.append(1)
result.append(1)
result.append(2)

try:
    for i in range(4, n+1):
        aux = result[i-1] + result[i-2] + result[i-3]
        result.append(aux)

except:
    print("An exception occurred")

print(result)
```

Ejecución

```
[MacBook-Pro-de-Aaron:p7 aarongarcia$ python3 index_1_1.py
Teclee el nombre del archivo: input_1.txt
[0, 1, 1, 2, 4, 7, 13]
```

Justificación

La complejidad de este algoritmo es $O(n-3)=O(n)$ ya que va tomando valores previos y los suma

Ejercicio 2

Dada una secuencia de números enteros a_1, a_2, \dots, a_n , encontrar la subsecuencia ascendente de longitud más larga. Por ejemplo, dados los enteros 5, 2, 8, 6, 3, 6, 9, 7 la subsecuencia más larga es 2, 3, 6, 9. Diseña un algoritmo por fuerza bruta para solucionar este problema. Escribe el pseudocódigo correspondiente y determina la complejidad de este. Posteriormente, diseña otro algoritmo que sea más eficiente que el anterior. Muestra que el nuevo algoritmo funciona y determina la complejidad de este.

Algoritmo por fuerza bruta (Recursivo)

Implementación

```
def recursive_solution(remaining_sequence, bigger_than=0):  
    """  
    Encuentra la subsecuencia creciente más larga de la secuencia  
    restante que es mayor bigger_than y la devuelve.  
    """  
  
    # Caso base: la secuencia es vacía  
    if len(remaining_sequence) == 0:  
        return remaining_sequence  
  
    # Caso recursivo 1: quitar el elemento actual y procesar el resto  
    best_sequence = recursive_solution(remaining_sequence[1:], bigger_than)  
  
    # Caso recursivo 2: incluimos el elemento actual si es lo suficientemente grande.  
    first = remaining_sequence[0]  
  
    if (first > bigger_than) or (bigger_than == 0):  
        sequence_with = [first] + recursive_solution(remaining_sequence[1:], first)  
        # Elejimos el caso 1 y el caso 2 que fueron más largos  
        if len(sequence_with) >= len(best_sequence):  
            best_sequence = sequence_with  
  
    return best_sequence
```

Ejecución

A partir de un archivo de texto se leen los datos necesarios (los números que contiene la secuencia original)

```
p7 > ≡ input_2.txt  
1    Lista de valores  
2    30 10 20 50 40 80 60|
```

```
[MacBook-Pro-de-Aaron:desktop aarongarcia$ python3 pruebas.py  
  
Lista original:  [5, 2, 8, 6, 3, 6, 9, 7]  
Subsecuencia máxima:  [2, 3, 6, 9]
```

Justificación

Dado que cada hay dos llamadas recursivas por cada N de la secuencia, entonces simplemente la complejidad es $O(2^n)$

Algoritmo eficiente

Implementación

```
# ----- Ejecutable
seq = leerDatos(name_file_INPUT)
subseq = subsequence(seq)
print("\nLista original: ", seq)
print("Subsecuencia máxima: ", subseq, "\n")
```

A partir de un archivo de texto se leen los datos necesarios (los números que contiene la secuencia original)

```
p7 > ≡ input_2.txt
1      Lista de valores
2      30 10 20 50 40 80 60
```

La función estrella es la siguiente:

```
def subsequence(seq):
    if not seq:
        return seq

    M = [None] * len(seq) # offset by 1 (j -> j-1)
    P = [None] * len(seq)

    # M es una lista. M [j-1] apuntará a un índice de seq que contiene el valor más pequeño
    # que podría usarse (al final) para construir una subsecuencia creciente de longitud j.
    # P es una lista. P [i] apuntará a M [j], donde i es el índice de seq. En pocas palabras,
    # indica cuál es el elemento anterior de la subsecuencia. P se utiliza para construir el resultado al final.

    # Como tenemos al menos un elemento en nuestra lista, podemos comenzar
    # sabiendo que hay al menos una subsecuencia creciente de longitud uno: el primer elemento
    # L es un número: se actualiza mientras recorre la secuencia y marca la longitud de la subsecuencia más larga
    L = 1
    M[0] = 0

    # Recorriendo la secuencia a partir del segundo elemento
    for i in range(1, len(seq)):
        # Búsqueda binaria: queremos la mayor j <= L
        # tal que seq [M [j]] < seq [i] (por defecto j = 0),
        # por lo tanto, queremos el límite inferior al final del proceso de búsqueda.
        lower = 0
        upper = L
```

```

# Dado que la búsqueda binaria no verá el valor del límite superior,
# tendremos que verificarlo manualmente
if seq[M[upper-1]] < seq[i]:
    j = upper

else:
    # bucle de búsqueda binaria real
    while upper - lower > 1:
        mid = (upper + lower) // 2
        if seq[M[mid-1]] < seq[i]:
            lower = mid
        else:
            upper = mid

    j = lower # esto también establecerá el valor predeterminado en 0

P[i] = M[j-1]

if j == L or seq[i] < seq[M[j]]:
    M[j] = i
    L = max(L, j+1)

# Creación del resultado: [seq [M [L-1]], seq [P [M [L-1]]], seq [P [P [M [L-1]]]], ...]
result = []
pos = M[L-1]
for _ in range(L):
    result.append(seq[pos])
    pos = P[pos]

return result[::-1] # invertir

```

Ejecución

```
[MacBook-Pro-de-Aaron:p7 aarongarcia$ python3 index_2.py  
Teclee el nombre del archivo: input_2.txt
```

```
Lista original:  [30, 10, 20, 50, 40, 80, 60]  
Subsecuencia máxima:  [10, 20, 40, 60]
```

```
[MacBook-Pro-de-Aaron:p7 aarongarcia$ python3 index_2.py  
Teclee el nombre del archivo: input_2.txt
```

```
Lista original:  [5, 2, 8, 6, 3, 6, 9, 7]  
Subsecuencia máxima:  [2, 3, 6, 7]
```

Justificación

Dado que utilizamos búsqueda binaria y es lo que mas peso tiene en el desarrollo de este programa, entonces la complejidad es la misma del algoritmo de búsqueda binaria, es decir, $O(\log n)$.