

	Carátula para entrega de prácticas	
Facultad de Ingeniería	Laboratorio de docencia	

Laboratorios de computación salas A y B

<i>Profesor:</i>	M. I. Marco Antonio Martínez Quintana
<i>Asignatura:</i>	Estructura de Datos y Algoritmos
<i>Grupo:</i>	17
<i>No de Práctica(s):</i>	11
<i>Integrante(s):</i>	Díaz Segura, Mauricio Iván
<i>No. de Equipo de cómputo empleado:</i>	
<i>No. de Lista o Brigada:</i>	
<i>Semestre:</i>	2020-2
<i>Fecha de entrega:</i>	23 - IV - 2020
<i>Observaciones:</i>	

CALIFICACIÓN: _____

OBJETIVO

El objetivo de esta guía es implementar, al menos, dos enfoques de diseño (estrategias) de algoritmos y analizar las implicaciones de cada uno de ellos.

INTRODUCCIÓN

Realizar un algoritmo no es un asunto de poca importancia, se requiere tener ideas ordenadas y de mucha organización. Para comenzar, y lo más importante, es siempre tener en mente las características de los algoritmos: debe ser preciso, definido, finito, correcto, tener una entrada y una salida, ser eficaz y producir un resultado.

A pesar de que pueda parecer lo contrario, realizar un algoritmo que cumpla todos estos rasgos no es algo simple o banal. Sin embargo, para llegar a un resultado bien hecho se pueden seguir varios caminos de desarrollo.

DESARROLLO Y RESULTADOS

La fuerza bruta es un método de lograr resultados eficaces, pero tardados. Cuando pruebas todas las posibilidades de un problema, hay mayores posibilidades de encontrar la solución al problema.

```
1 from string import ascii_letters , digits
2 from itertools import product
3
4 #Concatenar letras y dígitos en una sola cadena
5 caracteres = ascii_letters+digits
6
7 def buscador(con):
8
9     #Archivo con todas las combinaciones generadas
10    archivo = open("combinaciones.txt", "w")
11
12    if 3<= len(con) <= 4:
13        for i in range(3,5):
14            for comb in product(caracteres, repeat = i):
15                #Se utiliza join() para concatenar los caracteres regresados por
16                #la función product().
17                #Como join necesita una cadena inicial para hacer la
18                #concatenación, se pone una cadena vacía
19                #al principio
```

```

18         prueba = "".join(comb)
19         #Escribiendo al archivo cada combinaci n generada
20         archivo.write( prueba + "\n" )
21         if prueba == con:
22             print('Tu contrase a es {}'.format(prueba))
23             #Cerrando el archivo
24             archivo.close()
25             break
26     else:
27         print('Ingresa una contrase a que contenga de 3 a 4 caracteres')
28
29 from time import time
30 t0 = time()
31 con = 'H011'
32 buscador(con)
33 print("Tiempos de ejecuci n {}".format(round(time()-t0, 6)))

```

[Out] Tu contraseña es H011

Tiempos de ejecución 59.137606

Los algoritmos ávidos son bastante parecidos al método anterior, sólo que estos se definen por ser más ordenados. Suelen ser más rápidos, pero no siempre se obtiene el mejor resultado.

```

1 def cambio(cantidad, denominaciones):
2     resultado = []
3     while (cantidad > 0):
4         if (cantidad >= denominaciones[0]):
5
6             num = cantidad // denominaciones[0]
7             cantidad = cantidad - (num * denominaciones[0])
8             resultado.append([denominaciones[0], num])
9             denominaciones = denominaciones[1:] #Se va consumiendo la lista de
denominaciones
10         return resultado
11
12 #Pruebas del algoritmo
13 print (cambio(1000, [500, 200, 100, 50, 20, 5, 1]))
14 print (cambio(500, [500, 200, 100, 50, 20, 5, 1]))
15 print (cambio(300, [50, 20, 5, 1]))
16 print (cambio(200, [5]))
17 print (cambio(98, [50, 20, 5, 1]))

```

```
[Out] [[500, 2]]

      [[500, 1]]

      [[50, 6]]

      [[5, 40]]

      [[50, 1], [20, 2], [5, 1], [1, 3]]
```

Por otro lado, el *Bottom-up* se enfoca en resolver un problema a partir de información ya conocida. En el siguiente código, se introducen los primeros números de la serie de Fibonacci y, desde ahí, el programa comienza a realizar la seriación si el número ingresado es mayor a tres.

```
1 def fibonacci_bottom_up(numero):
2     f_parciales = [0, 1, 1] #Esta es la lista que mantiene las soluciones
    previamente calculadas
3     while len(f_parciales) < numero:
4         f_parciales.append(f_parciales[-1] + f_parciales[-2])
5         print(f_parciales)
6     return f_parciales[numero-1]
7
8 fibonacci_bottom_up(5)
```

```
[Out] 5
```

Al contrario, el *Top down* almacena todas las operaciones ya efectuadas para que no se repitan. Como en el caso siguiente, en donde, dentro de la variable `memoria` se guardan los números de la serie de Fibonacci para que no se calculen de nuevo.

```
1 def fibonacci_top_down(numero):
2     if numero in memoria: #Si el n mero ya se encuentra calculado, se
    regresa el valor ya ya no se hacen m s c lculos
3         return memoria[numero]
4     f = fibonacci_iterativo_v2(numero-1) + fibonacci_iterativo_v2(numero-2)
5     memoria[numero] = f
6     return memoria[numero]
7
8 fibonacci_top_down(12)
```

```
[Out] 89
```

De forma incremental, se revisa el algoritmo varias veces; en cada ocasión se modifica la información

hasta conseguir el objetivo. En el ejemplo siguiente, se acomoda una lista de números por orden ascendente. La operación se repite con cada número hasta lograr el orden esperado.

```
1 def insertionSort(n_lista):
2     for index in range(1, len(n_lista)):
3         actual = n_lista[index]
4         posicion = index
5         print("valor a ordenar = {}".format(actual))
6         while posicion > 0 and n_lista[posicion-1] > actual:
7             n_lista[posicion] = n_lista[posicion-1]
8             posicion = posicion - 1
9         n_lista[posicion] = actual
10        print(n_lista)
11        print()
12    return n_lista
13
14    lista = [15, 32, 4]
15    print("lista desordenada {}".format(lista))
16    insertionSort(lista)
17    print("lista ordenada {}".format(lista))
```

[Out] lista desordenada [15, 32, 4]

valor a ordenar = 32 [15, 32, 4]

valor a ordenar = 4 [4, 15, 32]

lista ordenada [4, 15, 32]

Divide y vencerás es una vía bastante aconsejable para lograr un buen resultado sin tener que romperse la cabeza. El problema general se divide en pequeñas secciones fáciles de resolver y, finalmente, se une todo para conseguir la salida deseada.

Por último, hay dos formas útiles de conocer un algoritmo. La primera es por tiempo, como se muestra en la siguiente imagen. Por medio de la librería `time`, la misma que usamos al principio de esta práctica, se puede contar el tiempo de ejecución de un programa y llevar el control del programa.

Por otro lado, también está el conteo, en donde se muestra la cantidad de veces que una función fue realizada.

```

In [36]: 1 #Tamaños de la Lista de números aleatorios a generar
          2 datos = [ii*100 for ii in range(1,21)]
          3
          4 tiempo_is = [] #Lista para guardar el tiempo de ejecución de insert sort
          5 tiempo_qs = [] #Lista para guardar el tiempo de ejecución de quick sort
          6
          7 for ii in datos:
          8     lista_is = random.sample(range(0, 10000000), ii)
          9     #Se hace una copia de la lista para que se ejecute el algoritmo con los mismo números
          10    lista_qs = lista_is.copy()
          11
          12    t0 = time() #Se guarda el tiempo inicial
          13    insertionSort(lista_is)
          14    tiempo_is.append(round(time()-t0, 6)) #Se le resta al tiempo actual, el tiempo inicial
          15
          16    t0 = time()
          17    quicksort(lista_qs)
          18    tiempo_qs.append(round(time()-t0, 6))

valor a ordenar = 26600
[26600, 2759242, 470661, 9405718, 8724359, 7288566, 3296493, 3489429, 6760316, 8262567, 1539497, 1239433, 5490516, 8965262, 55
61207, 6067057, 6402530, 3679971, 9687509, 1364414, 101603, 8945615, 6194908, 7961240, 5867939, 6267219, 8720439, 3549865, 730
5038, 3354651, 9199128, 1091423, 4973160, 826974, 7378961, 9654174, 4658805, 2163952, 8640631, 8066642, 1644107, 287700, 55010
57, 7528257, 7695938, 9493389, 5216900, 6734360, 320900, 7890323, 6795792, 1120455, 2106021, 4456022, 2589601, 1174062, 516944
0, 1083126, 2650886, 8234392, 3555310, 4986352, 9175388, 4287150, 1912818, 2922137, 6673840, 9241026, 4912753, 5838364, 91926
7, 6589907, 6017114, 4504000, 2298246, 2364423, 6962109, 5432902, 5407815, 7525417, 6045307, 4071460, 1874446, 4936780, 827786
0, 3063784, 5934777, 6581158, 6170071, 5260742, 8151242, 8931615, 9865454, 5253302, 556037, 3256039, 5479083, 8161778, 313281
8, 1073549]

valor a ordenar = 470661
[26600, 470661, 2759242, 9405718, 8724359, 7288566, 3296493, 3489429, 6760316, 8262567, 1539497, 1239433, 5490516, 8965262, 55
61207, 6067057, 6402530, 3679971, 9687509, 1364414, 101603, 8945615, 6194908, 7961240, 5867939, 6267219, 8720439, 3549865, 730
5038, 3354651, 9199128, 1091423, 4973160, 826974, 7378961, 9654174, 4658805, 2163952, 8640631, 8066642, 1644107, 287700, 55010
57, 7528257, 7695938, 9493389, 5216900, 6734360, 320900, 7890323, 6795792, 1120455, 2106021, 4456022, 2589601, 1174062, 516944
0, 1083126, 2650886, 8234392, 3555310, 4986352, 9175388, 4287150, 1912818, 2922137, 6673840, 9241026, 4912753, 5838364, 91926
7, 6589907, 6017114, 4504000, 2298246, 2364423, 6962109, 5432902, 5407815, 7525417, 6045307, 4071460, 1874446, 4936780, 827786
0, 3063784, 5934777, 6581158, 6170071, 5260742, 8151242, 8931615, 9865454, 5253302, 556037, 3256039, 5479083, 8161778, 313281
8, 1073549]

In [37]: 1 # Se imprimen los tiempos parciales de ejecución
          2 print("Tiempos parciales de ejecución en INSERT SORT {} [s] \n".format(tiempo_is))
          3 print("Tiempos parciales de ejecución en QUICK SORT {} [s]".format(tiempo_qs))

Tiempos parciales de ejecución en INSERT SORT [0.035003, 0.19169, 0.748782, 0.860984, 1.079015, 1.089005, 1.109232, 1.551001,
2.215106, 2.578381, 2.529001, 3.569953, 3.820005, 2.874445, 3.851767, 4.778711, 6.7708, 8.204188, 6.299482, 10.373555] [s]

Tiempos parciales de ejecución en QUICK SORT [0.041996, 0.361286, 0.490036, 0.840177, 0.9363, 1.042048, 1.241856, 1.749521, 1.9
92941, 1.914997, 2.535961, 2.545053, 2.511083, 2.97814, 3.659871, 3.474617, 4.243358, 5.448499, 3.978757, 12.09492] [s]

```

CONCLUSIONES

Realizar un algoritmo eficaz no es una cuestión simple, pero no hay que desesperarse por ello, ya que existen varios métodos para lograr una buena organización y conseguir el resultado que se busca.

REFERENCES

1. *Tutorial de Python*. Fecha de consulta: 20 de abril de 2020. Recuperado de <http://docs.python.org.ar/tutorial/3/real-index.html>
2. Jugaru, M. (2014). *Introducción a la programación*. México: Grupo Editorial Patria.