Análisis asintótico: algoritmos recursivos e iterativos

Frank Sebastián Franco Hernández

22 de agosto de 2014

1. Análisis de tres algoritmos de ordenamiento

1.1. Algoritmo BubbleSort

Este algoritmo funciona como las burbujas dentro del agua: dejando las más pesadas (mayores) en los últimos lugares del arreglo a ordenar y las más livianas (menores) en los primeros lugares. En principio el fundamento es simple: declara dos iteradores: uno para el array completo y otro que reorganizará el arreglo. Si lo que está indicado por el primer iterador es menor que lo que tiene el segundo, se mantienen lugares, y de lo contrario se intercambian.

El simple hecho de tener dos iteradores implica que el orden de este algoritmo es de $O(n^2)$.

1.2. Algoritmo Insertion Sort

Este algoritmo utiliza el mismo principio que cuando uno organiza cartas en una mano, organizando progresivamente las mismas de izquierda a derecha, de menor a mayor. Empieza con un iterador externo, fija un valor temporal y crea un nuevo iterador, que entrará siempre que se cumpla que el iterador sea mayor que cero y que el arreglo en el iterador sea mayor que el valor temporal. Una vez dentro de ese iterador, se asigna la posición en la que está, a la posición anterior y el temporal se pone en la posición en la que quedó. Como son dos iteradores, el orden sigue siendo de $O(n^2)$.

1.3. Algoritmo Merge Sort

Se trata de un algoritmo de dividir y vencer. En principio se promedian los índices inicial y final del arreglo, con el fin de enviar eso a la recursión. Ya con el array dividido, se sigue dividiendo hasta que queden arreglos de tamaño 1. Se ordena cada pedazo y se va fusionando asimismo el arreglo.

El hecho de dividir el arreglo por mitades, ya hace el algoritmo de orden logarítmico; pero como se iteran sobre esas divisiones, ya hace que sea de orden $O(n \log n)$.

2. Algoritmos en Python

2.1. Algoritmo Bubble Sort

```
__author__ = 'FrankSebastia'
import datetime
import random

def bubblesort( A ):
  for i in range( len( A ) ):
    for k in range( len( A ) - 1, i, -1 ):
      if ( A[k] < A[k - 1] ):
        tmp= A[k]
        A[k]=A[k-1]
        A[k-1]=tmp


for tamanio in range (10000):
    print tamanio
    lista=[]
    for x in range(tamanio):
        lista.append(random.randrange(1,10000))

    t1 = datetime.datetime.now()
    bubblesort(lista)
    t2 = datetime.datetime.now()
    print("Execution time: %s" %(t2-t1))
```

Los tiempos de ejecución de dicho algoritmo en un procesador Intel Core i3 (el que se está utilizando) son los de crecimiento más rápido. Con arreglos de tamaño 250 tarda 18 milisegundos, con 500 tarda 66 milisegundos (3 y dos tercios de vez más), con 1000 tarda 275 milisegundos (4.16 veces más), con 1500 tarda casi 1 segundo (916 milisegundos siendo exactos, unas 3.3 veces más) y con 2000 tardó casi 2 segundos (1,985 segundos; dos veces y un octavo más).

Sabemos que un crecimiento cuadrático implica que cuando se duplica el número que se tiene en la entrada, se cuadruplica la salida, lo cual se hace patente en este procedimiento.

2.2. Algoritmo Insertion Sort

```
__author__ = 'FrankSebastia'
import datetime
import random

def insertionsort( aList ):
  for i in range( 1, len( aList ) ):
    tmp = aList[i]
    k = i
    while k > 0 and tmp < aList[k - 1]:
        aList[k] = aList[k - 1]
        k -= 1
```

```
        aList[k] = tmp


for tamanio in range(10000):
    lista=[]
    for x in range(tamanio):
        lista.append(random.randrange(1,10000))

    t1 = datetime.datetime.now()
    insertionsort(lista)
    t2 = datetime.datetime.now()
    print tamanio
    print("Execution time: %s" %(t2-t1))
```

Aquí ya empieza a mejorar un poco la situación, hablando de tiempos de ejecución. Con el procesador utilizado, con arreglos de tamaño 250 tardó 16 milisegundos, con 500 tardó 53 milisegundos (poco más de 3 veces más), con 1000 tardó 271 milisegundos (poco más de 5 veces más) y con 2000 tardó poco más de 1 segundo (1,021 s), teniendo en 1500 un tiempo cercano a las 500 milésimas de segundo (0,542 s).

En este caso, el algoritmo, aunque pareciera tener un comportamiento marcadamente cuadrático al principio, tiende a estabilizarse.

2.3. Algoritmo Merge Sort

```
__author__ = 'FrankSebastia'
import datetime
import random

def mergesort( aList ):
  _mergesort( aList, 0, len( aList ) - 1 )


def _mergesort( aList, first, last ):
  # break problem into smaller structurally identical pieces
  mid = ( first + last ) / 2
  if first < last:
    _mergesort( aList, first, mid )
    _mergesort( aList, mid + 1, last )

  # merge solved pieces to get solution to original problem
  a, f, l = 0, first, mid + 1
  tmp = [None] * ( last - first + 1 )

  while f <= mid and l <= last:
    if aList[f] < aList[l] :
      tmp[a] = aList[f]
      f += 1
    else:
      tmp[a] = aList[l]
      l += 1
```

```
    a += 1

  if f <= mid :
    tmp[a:] = aList[f:mid + 1]

  if l <= last:
    tmp[a:] = aList[l:last + 1]

  a = 0
  while first <= last:
    aList[first] = tmp[a]
    first += 1
    a += 1

for tamanio in range (10000):
    print tamanio
    lista=[]
    for x in range(tamanio):
        lista.append(random.randrange(1,10000))

    t1 = datetime.datetime.now()
    mergesort(lista)
    t2 = datetime.datetime.now()

    print("Execution time: %s" %(t2-t1))
```

Es sin duda, el más rápido de los tres y es quien organizó los 10000 arreglos en un tiempo razonable. La magnitud de la rapidez de este algoritmo, comparándolo con los otros dos, es que un arreglo de tamaño en el rango de los 9000 lo organiza en tiempos en los que los otros dos organizarían apenas en el rango de los 1000, haciéndole aproximadamente 9 o más veces más rápido que los otros dos.

3. Multiplicación dentro de un arreglo

```
__author__ = 'FrankSebastia'

def multi(a, x): #a es una lista, x es un valor entero
    for i in range(len(a)):
        if x%a[i]==0 and x/a[i] in a: # el operador in pregunta si existe tal e
            print "Los numeros de este arreglo que multiplicados dan "+x + " so
        else:
            return "No hay numeros que satisfagan la condicion"
```

Recibe una lista de tamaño cualquiera y un valor entero cualquiera. Itera sobre la lista preguntando si el módulo entre el valor entero y el elemento i-ésimo de la lista es cero y si el cociente entre el entero y ese mismo valor existe en la lista (Python provee un operador llamado *in* que permite hacer esto en una sola línea). Con ambas condiciones verdaderas, el algoritmo devuelve por pantalla el resultado. De lo contrario, imprime que no existe número alguno dentro del arreglo que satisfaga la condición.

4. Análisis de un algoritmo recursivo

Con el procesador utilizado, el algoritmo terminaba con errores de desbordamiento de pila (o directamente se moría) en cualquier lenguaje (Python, Java o C++), así que se optó por el uso de Excel para hacer dicha función, porque se optó por imprimir todos los casos posibles. El promedio de llamadas para cada número es de casi 87 (86,9664), siendo el menor cuando apenas inicia (1) y el mayor situándose en 9225, teniendo 259 llamadas recursivas para llegar al caso base. La cantidad de llamadas por cada número es demasiado fluctuante, no sigue un patrón en específico, aunque se sabe al menos que los números pares requieren de menos llamadas recursivas que los impares. Es impracticable poner la tabla y el gráfico porque son de enormes dimensiones, por lo que se incluyen en archivo adjunto, llamado "función.xlsx".