Ordenar cuadráticamente

Introducción

Ya platicamos acerca de la importancia del problema de ordenar y nos pusimos de acuerdo en ciertas sutilezas. También discutimos propiedades deseables que pueden tener los algoritmos de ordenación.

En esta entrada veremos varios posibles algoritmos mediante los cuales podemos ordenar una lista de números. Todos estos algoritmos se ejecutan en tiempo cuadrático en el tamaño de la entrada. Sin embargo, las ideas que usan y los detalles de qué sucede en cada uno van variando.

Para simplificar la discusión, daremos principalmente ideas para ordenar números en orden creciente. Pero las mismas ideas funcionan para otros órdenes parciales.

SelectionSort

El algoritmo `SelectionSort` tiene una idea intuitiva muy simple. Si comenzamos con una lista de objetos $L=[a_1,a_2,\ldots,a_n]$ y los queremos ordenar, entonces simplemente identificamos en la lista al menor elemento, lo quitamos de L y lo ponemos al final de una lista M. Repetimos esto hasta agotar todos los elementos de L. El algoritmo es correcto pues se agotan todos los elementos de L y se van poniendo en orden en M. Una demostración más formal podría hacerse por inducción: en el paso inductivo entra un nuevo elemento a M y es mayor que todos los anteriores pues lo elegimos después que todos los anteriores.

Una manera de hacer esta implementación es la siguiente. Primero, necesitamos una subrutina que identifique al menor elemento de una lista.

```
def encontrar_minimo(L):
# Recibe una lista no vacía L y regresa su mínimo
    minimo=L[0]
    for j in L:
        if j<minimo:
            minimo=j
    return minimo

print(encontrar_minimo([2,6,4,7]))
print(encontrar_minimo([2,-3,8]))
print(encontrar_minimo([9,11,0,6,4,7]))
```

```
2
-3
0
```

Esta subrutina para encontrar el mínimo toma tiempo $O(n)$ (aquí n es la cantidad de elementos de la lista) pues el ciclo recorre todos los elementos de L. Usamos `encontrar_minimo` para implementar `SelectionSort` a continuación.

```
def selection_sort(L):
# Recibe una lista no vacía L y la regresa ordenada.
    M=[] #La lista que iremos llenando en orden
    while L!=[]:
        minimo=encontrar_minimo(L)
        L.remove(minimo)
        M.append(minimo)
    return(M)

print(selection_sort([2,6,4,7]))
print(selection_sort([2,-3,8,2]))
print(selection_sort([9,11,0,6,4,7,4]))
```

```
[2, 4, 6, 7]
[-3, 2, 2, 8]
[0, 4, 4, 6, 7, 9, 11]
```

Aquí el bucle `while` toma n iteraciones, una por cada elemento de M. Dentro de dicho bucle estamos llamando la función `encontrar_minimo` que es de tiempo $O(n)$. También estamos eliminando al elemento mínimo de L, (que como mucho tomaría tiempo $O(n)$) y agregándolo a M, (que como mucho tomaría tiempo $O(n)$). Así, la cantidad de tiempo total es

$$O(n)(O(n)+O(n)+O(n))$$
que se simplifica a $O(n^2)$.

Aquí estamos creando una nueva lista M que tendrá $O(n)$ elementos. Dependiendo de exactamente cómo vayamos pasando los elementos de L a M, es posible que necesitemos de $O(n)$ espacio adicional (si preservamos L) o no (si vamos liberando de memoria sus elementos).

A continuación hay una «implementación didáctica» de `SelectionSort` con un código un poco más largo, pero que puede ayudar a entender qué es lo que va sucediendo cada vez que se hace el bucle. Como salida, el algoritmo va diciendo las decisiones que se van tomando y por qué se pasa cierto elemento de L a M. Observa que aunque estemos haciendo más cosas, de todas formas corre en tiempo $O(n^2)$, pues los reportes que va haciendo toman tiempo constante.

```
def selection_sort_amigable(L):
# Recibe una lista no vacía L y la regresa ordenada.
    M=[] #La lista que iremos llenando en orden
    paso=1
    print("Comenzamos con la lista L como sigue: {}\n ----".format(L))
    while L!=[]:
        minimo=encontrar_minimo(L)
        print("Paso {}\n ----".format(paso))
        print("Ahora la lista L es {}, que tiene mínimo {}".format(L,minimo))
        print("La lista M es {}".format(M))
        print("Así, tenemos que pasar el número {} de L a M".format(minimo))
        print("")
        L.remove(minimo)
        M.append(minimo)
        paso+=1
    print("Esto ya dejó a L vacía. De esta manera, la lista ya en orden es M={}".format(M))

selection_sort_amigable([9,0,6,4,7,4])
```

```
Comenzamos con la lista L como sigue: [9, 0, 6, 4, 7, 4]
 ----
Paso 1
 ----
Ahora la lista L es [9, 0, 6, 4, 7, 4], que tiene mínimo 0
La lista M es []
Así, tenemos que pasar el número 0 de L a M

Paso 2
 ----
Ahora la lista L es [9, 6, 4, 7, 4], que tiene mínimo 4
La lista M es [0]
Así, tenemos que pasar el número 4 de L a M

Paso 3
 ----
Ahora la lista L es [9, 6, 7, 4], que tiene mínimo 4
La lista M es [0, 4]
Así, tenemos que pasar el número 4 de L a M

Paso 4
 ----
Ahora la lista L es [9, 6, 7], que tiene mínimo 6
La lista M es [0, 4, 4]
Así, tenemos que pasar el número 6 de L a M

Paso 5
 ----
Ahora la lista L es [9, 7], que tiene mínimo 7
La lista M es [0, 4, 4, 6]
Así, tenemos que pasar el número 7 de L a M

Paso 6
 ----
Ahora la lista L es [9], que tiene mínimo 9
La lista M es [0, 4, 4, 6, 7]
Así, tenemos que pasar el número 9 de L a M

Esto ya dejó a L vacía. De esta manera, la lista ya en orden es M=[0, 4, 4, 6, 7, 9]
```

InsertionSort

La idea de `InsertionSort` también es muy sencilla. De nuevo tenemos como entrada una lista L y usaremos una lista auxiliar M. Lo que haremos es tomar un elemento de L y colocarlo «en el lugar correcto» de M. Para ello, iremos preguntándonos de izquierda a derecha si el elemento que tenemos puede entrar en cierta posición de M, o le toca una más a la derecha.

Veamos esto en acción con la lista inicial

$$L=[2,8,3,2,9].$$
El primer elemento de L, de izquierda a derecha es 2. Así, este será el primer elemento que ponemos en M:

$$M=[2].$$
El segundo elemento de L es 8. ¿Va en la primer posición de M? No, pues el 2 es menor o igual. ¿Quedan más elementos para compararlo? No. Entonces va hasta el final. Esto deja

$$M=[2,8].$$
El tercer elemento de L es 3. ¿Va en la primer posición de M? No, pues el 2 es menor o igual. ¿Va en la segunda? Sí, pues el 3 es más grande. Entonces hasta ahora

$$M=[2,3,8].$$
El cuarto elemento de L es 2. ¿Va en la primer posición de M? No, pues el 2 es menor o igual. ¿Va en la segunda? Sí, pues el 3 es más grande. En este punto

$$M=[2,2,3,8].$$
Finalmente, el último elemento de L es 9. ¿Va en la primer posición de M? No, pues el 2 es menor o igual. ¿En la segunda? No, pues el 2 es menor o igual. ¿En la tercera? No, pues el 3 es menor o igual. ¿En la cuarta? No, pues el 8 es menor o igual. ¿Quedan más elementos? No, entonces va hasta el final. Así, llegamos a la respuesta final

$$M=[2,2,3,8,9].$$
El algoritmo es correcto pues se toman todos los elementos de L en algún momento. Además, al ponerlos en M estamos garantizando que todos a la izquierda sean menores y todos a la derecha mayores por la transitividad de la desigualdad.

En términos de eficiencia, recorremos $O(n)$ elementos de L. Puede que tengamos suerte y cada elemento lo coloquemos tan pronto como sea posible (en 1 paso), pero en ocasiones tendrán que irse hasta la derecha y entonces usaremos $O(n)$ pasos. Así, el algoritmo corre en tiempo $O(n^2)$. Una vez más, podemos ir liberando de memoria los elementos de L para sólo usar $O(1)$ espacio extra, o bien preservarlo y necesitaríamos $O(n)$ espacio extra.

BubbleSort

Un último algoritmo cuadrático que discutiremos es `BubbleSort`. Ahora la idea está expresada mediante el siguiente pseudocódigo:

```
definimos BubbleSort(L):
    recorremos L de izquierda a derecha, checando pares adyacentes
       si todos los pares adyacentes están bien:
          la lista L ya está ordenada
       si algún par adyacente está mal:
          intercambiamos a los elementos del par
    repetimos y repetimos hasta que ya quede ordenada
```

Es posible que necesitemos recorrer la lista más de una vez para que quede ordenada. Por ejemplo consideremos la lista

$$[4,3,1,2].$$
El primer par es $4,3$, que está mal, así que lo intercambiamos para obtener

$$[3,4,1,2].$$
El segundo par es $4,1$ que está mal, así que lo cambiamos para obtener

$$[3,1,4,2].$$

El tercer par es \(4,2\) y está mal, así que lo cambiamos para obtener

\[[3,1,2,4].\]
Aquí llegamos al final de la lista y aún no se ordena. Entonces, volvemos a empezar: el primer par es \(3,1\) que está mal e intercambiamos para obtener

\[[1,3,2,4].\]
El segundo par es \(3,2\) que está mal e intercambiamos para obtener

\[[1,2,3,4].\]
El tercer par es \(3,4\) y ya está bien. Al dar una última pasada a la lista, vemos que ya no hay que cambiar pares y por lo tanto la lista ya queda bien.

Aquí no es totalmente claro que el algoritmo debería terminar. ¿Qué es lo que nos garantiza que no estaremos revolviendo los elementos una y otra vez?

Una forma de mostrar la correctitud es observando que tras el primer recorrido completo el elemento más grande se «arrastra» o «burbujea» de izquierda a derecha (por eso `BubbleSort`). Tras el segundo recorrido, el penúlimo se «arrastra» a la penúltima posición. Y así sucesivamente. Una prueba formal podría hacerse mostrando inductivamente para cada \(k\) de \(1\) a \(n\) que tras \(k\) recorridos completos por la lista, los últimos \(k\) elementos quedarán en orden.

Esto último nos dice que quizás tengamos que hacer \(O(n)\) recorridos completos, cada uno de los cuales toma \(O(n)\) pasos. De modo que `BubbleSort` también corre en tiempo cuadrático. ¿Cómo es en espacio?

Observa la siguiente implementación didáctica para entender el algoritmo todavía más:

```
def bubble_sort_amigable(L):
# Recibe una lista no vacía L y la ordena
    n=len(L)
    for j in range(n):
        print("Comienza el recorrido completo {}.\n----".format(j+1))
        for j in range(n-1):
            print("La lista ahora es {} y estamos checando la pareja {}, {}.".format(L,L[j],L[j+1]))
            if L[j]>L[j+1]:
                L[j],L[j+1]=L[j+1],L[j]
                print("Como está mal, la intercambiamos para obtener L={}.".format(L))
            else:
                print("Como está bien, no hacemos nada.")
        print("----\n")
    print("La lista ordenada es L={}".format(L))

bubble_sort_amigable([9,0,6,4,-1])
```

```
Comienza el recorrido completo 1.
----
La lista ahora es [9, 0, 6, 4, -1] y estamos checando la pareja 9, 0.
Como está mal, la intercambiamos para obtener L=[0, 9, 6, 4, -1].
La lista ahora es [0, 9, 6, 4, -1] y estamos checando la pareja 9, 6.
Como está mal, la intercambiamos para obtener L=[0, 6, 9, 4, -1].
La lista ahora es [0, 6, 9, 4, -1] y estamos checando la pareja 9, 4.
Como está mal, la intercambiamos para obtener L=[0, 6, 4, 9, -1].
La lista ahora es [0, 6, 4, 9, -1] y estamos checando la pareja 9, -1.
Como está mal, la intercambiamos para obtener L=[0, 6, 4, -1, 9].
----

Comienza el recorrido completo 2.
----
La lista ahora es [0, 6, 4, -1, 9] y estamos checando la pareja 0, 6.
Como está bien, no hacemos nada.
La lista ahora es [0, 6, 4, -1, 9] y estamos checando la pareja 6, 4.
Como está mal, la intercambiamos para obtener L=[0, 4, 6, -1, 9].
La lista ahora es [0, 4, 6, -1, 9] y estamos checando la pareja 6, -1.
Como está mal, la intercambiamos para obtener L=[0, 4, -1, 6, 9].
La lista ahora es [0, 4, -1, 6, 9] y estamos checando la pareja 6, 9.
Como está bien, no hacemos nada.
----

Comienza el recorrido completo 3.
----
La lista ahora es [0, 4, -1, 6, 9] y estamos checando la pareja 0, 4.
Como está bien, no hacemos nada.
La lista ahora es [0, 4, -1, 6, 9] y estamos checando la pareja 4, -1.
Como está mal, la intercambiamos para obtener L=[0, -1, 4, 6, 9].
La lista ahora es [0, -1, 4, 6, 9] y estamos checando la pareja 4, 6.
Como está bien, no hacemos nada.
La lista ahora es [0, -1, 4, 6, 9] y estamos checando la pareja 6, 9.
```

```
Como está bien, no hacemos nada.
----

Comienza el recorrido completo 4.
----
La lista ahora es [0, -1, 4, 6, 9] y estamos checando la pareja 0, -1.
Como está mal, la intercambiamos para obtener L=[-1, 0, 4, 6, 9].
La lista ahora es [-1, 0, 4, 6, 9] y estamos checando la pareja 0, 4.
Como está bien, no hacemos nada.
La lista ahora es [-1, 0, 4, 6, 9] y estamos checando la pareja 4, 6.
Como está bien, no hacemos nada.
La lista ahora es [-1, 0, 4, 6, 9] y estamos checando la pareja 6, 9.
Como está bien, no hacemos nada.
----

Comienza el recorrido completo 5.
----
La lista ahora es [-1, 0, 4, 6, 9] y estamos checando la pareja -1, 0.
Como está bien, no hacemos nada.
La lista ahora es [-1, 0, 4, 6, 9] y estamos checando la pareja 0, 4.
Como está bien, no hacemos nada.
La lista ahora es [-1, 0, 4, 6, 9] y estamos checando la pareja 4, 6.
Como está bien, no hacemos nada.
La lista ahora es [-1, 0, 4, 6, 9] y estamos checando la pareja 6, 9.
Como está bien, no hacemos nada.
----

La lista ordenada es L=[-1, 0, 4, 6, 9]
```

Tarea moral

Los siguientes problemas te ayudarán a practicar lo visto en esta entrada. Para resolverlos, necesitarás usar herramientas matemáticas, computacionales o ambas.

1. Aplica manualmente `SelectionSort`, `InsertionSort` y `BubbleSort` a la siguiente lista

   \[31,45,46,51,32,17,54,36,21,22,31,36.\]

   Para esta lista, ¿cuál de los algoritmos te parece «mejor»? Explica tus criterios para decidir.

2. Explora el siguiente interactivo de VisuAlgo para ver más ejemplos de cómo funcionan `SelectionSort`, `InsertionSort` y `BubbleSort`: https://visualgo.net/en/sorting. En la parte superior puedes elegir el algoritmo a visualizar.
3. Demuestra formalmente que el peor caso de `InsertionSort` ocurre cuando la lista de entrada ya está ordenada y todos los elementos son distintos entre sí. Luego, haz una «implementación didáctica» de `InsertionSort`.
4. Encuentra de manera explícita los mejores y peores casos de tiempo de ejecución para el algoritmo `BubbleSort`.
5. Implementa un algoritmo de tipo `BubbleSort` en Python que te permita ordenar las siguientes parejas de números de acuerdo a su segunda coordenada:

   \[(4,1),(5,21),(3,12),(23,12),(11,14),(12,17),(21,18),(5,7),(7,5),(9,41),(11,2),(2,4).\]

Anterior
El problema de ordenar

Siguiente
Estructuras de datos vs. tipos de dato abstractos

Por Leonardo Ignacio Martínez Sandoval