Análisis de Algoritmos

Parte 1: análisis gráfico

Ordenar

- Frecuentemente queremos revisar una lista de valores, para lo cual aveces es conveniente ordenarlos.

- Utilizaremos un par de algoritmos de ordenamiento (inserción y mezclas) para ver su comportamiento gráficamente.

Inserción

- La idea es similar a como se ordena una mano de cartas:
- Si ya tenemos ordenadas las cartas `[2, 4, 5, 10]` y hasta la derecha tenemos un `7`, este buscará un lugar a la izquierda donde insertarse, en este caso el `10` le hace un espacio a la derecha pues es mas grande, así el `7` se sitúa entre el `5` y el `10`.

Más a detalle…

Si A es la lista de números a ordenar, **`A[0]`** es el primer elemento y **`len(A)`** es el número de elementos de **`A`**, entonces:

1. `Para j de 1 a len(A)-1`
2. `  pivote = A[j]` # carta a acomodar
3. `  i = j - 1` # posición a la izquierda del pivote
4. `  Mientras i >= 0 y A[i] > pivote`
5. `    A[i+1] = A[i]`
6. `    i = i - 1`
7. `  A[i + 1] = pivote`

Ejercicio

- En el sitio del curso [http://github.com/fhca] revisar el programa insercion.py y el programa pruebas_insercion0.py

- En insercion.py está implementado el algoritmo anterior en el lenguaje Python de dos maneras: **`ordenar()`**, que simplemente realiza las operaciones, y en **`ordenar2()`** que además tiene al final de cada expresión un “contador de pasos”

- En pruebas_insercion0.py se utiliza el programa anterior corriéndolo 1000 veces por cada tipo de entrada, guardando en archivos las parejas **`(n, pasos)`** para indicar el número de pasos que usa el programa para listas de números (entradas) de tamaño n

Análisis de Algoritmos

Un algoritmo lo podemos analizar por los siguientes aspectos:

- La **forma de la entrada** (para un tamaño de entrada fijo):
    - Mejor caso (entrada que hace que el algoritmo se tarde lo menos posible)
    - Caso promedio (entrada promedio, al azar)
    - Peor caso (entrada que hace que el algoritmo se tarde lo más posible)
- El **tamaño de la entrada**
    - Tiempo (número de pasos)
    - Espacio (cantidad de memoria)

Forma de la entrada

Por la forma de la entrada, el algoritmo de inserción presenta los siguientes casos:

- **Mejor caso**: cuando la lista ya está ordenada. En este caso, se revisa cada pivote, pero el ciclo while no se ejecuta, pues cada pivote ya está en su lugar

- **Caso promedio**: Cuando la lista es de números sin ningún orden aparente (desordenados)

- **Peor caso**: Cuando la lista está ordenada al revés, es decir, de mayor a menor. Esto es lo peor, pues cada pivote es más grande que los `j-1` números a su izquierda. Estos se desplazan un lugar a la derecha (con el `while`) para situar a cada pivote al principio de la lista.

- Nota que no para todos los algoritmos tiene que ser esto así.

Tamaño de la entrada

- Por lo general utilizaremos n para representar el tamaño de la entrada, en este caso `n = len(A)`, o sea, es la longitud de la lista `A`.

- Habría que correr pruebas_insercion.py para verlo, pero una revisión del algoritmo nos dice que la `j` se mueve `n-1` veces, esto es, hay `n-1` pivotes y las líneas posteriores a la 1 se repetirán `n-1` veces.

- En particular las líneas 2 y 3 se ejecutarán una vez por cada vuelta de la `j`

- El ciclo `Mientras` que empieza en la linea 4, en el mejor caso no se ejecutará nunca, en el peor caso se ejecutará `n(n-1)/2` veces y en el caso promedio, se ejecutará aproximadamente la mitad de esto.

- Por qué `n(n-1)/2`?. Ese ciclo `Mientras`, para `n` fija, se ejecuta para `j=1`, una vez; para `j=2`, dos veces; …; para `j=n-1`, `n-1` veces. Así se ejecutará `1 + 2 + … + (n - 1) = n(n-1)/2` veces, esto es aprox. n^2 veces.

Tamaño de la entrada (cont.)

- Por otra parte, todo el ordenamiento de inserción se hace sobre la misma lista A (se dice que se hace "*in situ*"), salvo por el espacio que ocupa el pivote y las variables **`i, j`**, esto es, un espacio de **`n+3`**, o sea, aprox. **`n`** unidades de espacio.

Ordenar por mezclas

- Utiliza el método de solución llamado “divide y vencerás”, en el que el problema se subdivide en problemas de menor tamaño que se resuelven recursivamente y cuyas soluciones se mezclan para formar la solución total.

- Si `A=[4, 2, 7, 5, 2, 3, 1, 6]` es la lista de números que queremos ordenar,

- primero dividimos `A` en dos sublistas, la izquierda `L=[4, 2, 7, 5]` y la derecha `R=[2, 3, 1, 6]`

- procedemos a ordenar recursivamente estas sublistas, resultado que ahora `L=[2, 4, 5, 7]` y `R=[1, 2, 3, 6]`

- a continuación viene la parte más importante: la lista ordenada final resulta de la mezcla de `L` y R

- para ello vaciamos los elementos de las sublistas `L=[2, 4, 5, 7]` y `R=[1, 2, 3, 6]` en `A`, de manera que en `A` se vaciará primero el menor elemento de las cabezas de cada sublista en cada ocasión.

- Así primero se compara el `2` de `L` con el `1` de `R`, ganando el `1`. Luego se comparan el mismo `2` de `L` con el `2` de `R`, resultando en empate (en este caso siempre se escoge el valor de `L` para desempatar), así `A = [1, 2]`. Luego comparamos el `4` de `L` y el `2` de `R`, ganando el `2`. Luego van el `4` de `L` y el `3` de `R`, ganando el `3`. En este punto `A` ya tiene `[1, 2, 2, 3]`. Luego comparamos el `4` de `L` y el `6` de `R`, ganando el `4`. Luego van el `5` de `L` y el `6` de `R`, ganando el `5`. Hasta aquí `A = [1, 2, 2, 3, 4, 5]`. Luego comparamos el `7` de `L` con el `6` de `R`, ganando el `6`, y finalmente ponemos en `A` el elemento (o elementos en caso de ser varios) que restan en `L`, resultando que `A= [1, 2, 2, 3, 4, 5, 6, 7]`

- Con ello tendremos la lista `A` ordenada.
- Por supuesto, lo anterior no es más que el “rastreo” del algoritmo para un ejemplo fijo, y no el algoritmo en sí.
- El algoritmo en sí comienza describiendo el procedimiento de mezcla de las sublistas `L` y `R`. Aquí `L` es la sublista de `A` que va de `p` a `q`, y `R` es la que va de `q+1` a `r:`

```
mezcla(A, p, q, r):
 1.  n1 = q - p + 1
 2.  n2 = r - q
 3.  Sean L[0..n1-1] y R[0..n2-1] las nuevas sublistas
 4.  Para i=0 hasta n1-1
 5.    L[i]=A[p+i]
 6.  Para j=0 hasta n2-1
 7.    R[j]=A[q+j+1]
 8.  L[n1]=∞
 9.  R[n2]=∞
10.  i=0
11.  j=0
12.  Para k=p hasta r
13.    Si L[i]<=R[j
14.      A[k]=L[i]
15.      i=i+1
16.    de lo contrario: A[k]=R[j]
17.      j=j+1
```

- Después de inicializar las sublistas L y R, ponemos el símbolo infinito (un número muy grande) a fin de que un número normal comparado con este sea menor (líneas 8 y 9).

- En las líneas 12-17 se acomoda en **`A[k]`** el menor elemento **`L[i]`** o **`R[j]`**, incrementándose los valores de **`i`** o de **`j`** según corresponda.

- A continuación escribimos el procedimiento que resta, es decir, aquél que realiza la subdivisión y se aplica recursivamente sobre las sublistas y por último realiza el procedimiento **`merge()`** anterior

```
merge-sort(A, p, r):
1.  Si p<r:
2.     q=(p+r)//2  # en Python "//" es división entera
3.     merge-sort(A, p, q)
4.     merge-sort(A, q+1, r)
5.     merge(A, p, q, r)
```

Para la primera vez, este procedimiento se llamará con **`merge-sort(A, 0, len(A)-1)`**

Ejercicio. Revisar el programa merge.py en el sitio web.

Análisis geométrico de algoritmos

- Una primera aproximación al análisis del tiempo que tardan los algoritmos inserción y mezclas viene dado por los programas pruebas-insercion0.py y pruebas-merge0.py.

- Estos generan archivos para el mejor, peor y caso promedio.

- Utilizan los procedimientos **`ordenar2()`** de los programas insercion.py y merge-sort.py respectivamente.

- Este procedimiento en ambos programas incrementa el contador de pasos en uno por cada expresión del programa (de momento a eso le llamaremos “paso”).

- Así los programas **`…0.py`** iteran el valor de **`n`** desde 1 hasta 1000, creando un arreglo especial para cada caso (mejor, promedio, peor), resetean el contador de pasos y escriben este junto con **`n`** en un archivo de texto ASCII que posteriormente se puede leer desde una hoja de cálculo o programa graficador como gnuplot. Así este archivo contendrá parejas como estas:

1	0
2	6
3	15
4	27
5	42
6	60
7	81
8	105
9	132
10	162
11	195
12	231
13	270
14	312
15	357
16	405
17	456
18	510
19	567
20	627
.	. .

- En el caso de pruebas-merge0.py, además generará tres archivos donde en vez de medir el número de pasos, mide el tiempo en segundos que tarda cada ejecución de **`ordenar()`** de tamaño **`n`**.

- Las gráficas del algoritmo de inserción corresponden a una parábola (n^2), mientras que las de merge-sort parecen una recta, pero tiene en realidad una pequeña curva (nlogn).