2018-2 I1
3. Ordenación

Una observación que hicimos en clase sobre los algoritmos de ***ordenación por comparación de ele-mentos adyacentes***, p.ej., *insertionSort*( ), es que su debilidad (en términos del número de operacio-nes que ejecutan) radica justamente en que sólo comparan e intercambian elementos adyacentes. Así, si tuviéramos un algoritmo que usara la misma estrategia de *insertionSort*( ), pero que comparara elementos que están a una distancia > 1 entre ellos, entonces podríamos esperar un mejor desempeño.

a) Calcula cuántas comparaciones entre elementos hace *insertionSort*( ) para ordenar el siguiente arreglo **a** de menor a mayor; muestra que entiendes cómo funciona *insertionSort*( ): **a** = **[**11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1**]**.

insertionSort( ) coloca el segundo elemento ordenado con respecto al primero, luego el tercero ordena-do con respecto a los dos primeros (ya ordenados entre ellos), luego el cuarto ordenado con respecto a los tres primeros (ya ordenados entre ellos), etc. En el caso del arreglo **a**, *insertionSort*( ) básicamente va moviendo cada elemento, 10, 9, …, 1, hasta la primera posición del arreglo. Para ello, el 10 es comparado una vez (con el 11), el 9 es comparado dos veces (con el 11 y con el 10), el 8 es comparado tres veces (con el 11, el 10 y el 9), y así sucesivamente; finalmente, el 1 es comparado 10 veces. Luego el total de comparaciones es 1 + 2 + 3 + ... + 10 = **55**.

b) Calcula ahora cuántas comparaciones entre elementos hace el siguiente algoritmo *shellSort*( ) para ordenar el mismo arreglo **a**. Muestra que entiendes cómo funciona *shellSort*( ); en particular, ¿qué relación tiene con *insertionSort*( )?

```
shellSort(a):
        gaps[] = {5,3,1}
        t = 0
        while t < 3:
           gap = gaps[t]
           j = gap
           while j < a.length]:
              tmp = a[j]
              k = j
              while k >= gap and tmp < a[k-gap]:
                 a[k] = a[k-gap]
                 k = k-gap
              a[k] = tmp
              j = j+1
        t = t+1
```

*Notemos que las comparaciones entre elementos de **a** se dan sólo en la comparación **tmp < a[k-gap]**; el algoritmo realiza **11** de estas comparaciones con resultado **true** y otras **17** con resultado **false**; en total, **28**.*

*Primero, realiza insertionSort entre elementos que están a distancia 5 entre ellos (según las posiciones que ocupan en **a**, no en cuanto a sus valores): el 6 con respecto al 11, el 5 c/r al 10, el 4 c/r al 9, el 3 c/r al 8, el 2 c/r al 7, el 1 c/r al 11, y el 1 c/r al 6.*

Luego, realiza insertionSort entre elementos que están a distancia 3 (nuevamente, según sus posiciones en el arreglo): el 2 c/r al 5 y el 7 c/r al 10.

Finalmente, realiza insertionSort entre elementos que están a distancia 1: el 3 c/r al 4 y el 8 c/r al 9; estos son los dos únicos pares de valores que aún están "desordenados" al finalizar el paso anterior.

c) **[**Independiente de a) y b)**]** Tenemos una lista de N números enteros positivos, ceros y negativos. Queremos determinar cuántos tríos de números suman 0. Da una forma de resolver este problema con complejidad mejor que $O(N^3)$.

Se puede hacer en tiempo $O(n^2 \log n)$: Primero, ordenamos la lista de menor a mayor, en tiempo $O(n \log n)$. Luego, para cada par de números, sumamos los dos números y buscamos en la lista ya ordenada, emplean-do búsqueda binaria, un número que sea el negativo de la suma; si lo encontramos, entonces incrementamos el contador de los tríos que suman 0. Hay $O(n^2)$ pares (los podemos generar sistemáticamente con dos loops, uno anidado en el otro) y cada búsqueda binaria se puede hacer en tiempo $O(\log n)$.

2018-2 EXAMEN

2. **Ordenación**

Queremos ordenar n datos, que vienen en un arreglo a, en que cada dato es un número entero en el rango 0 a k. Considera el siguiente algoritmo:

```
countSort(a, n, k):
    sea count[0 ... k] un arreglo de enteros
    for i = 0 ... k: count[i] = 0
    for j = 0 ... n–1: count[a[j]] = count[a[j]] + 1
    for i = 0 ... k:
        if count[i] > 0:
            for p = 1 ... count[i]: print(i)
```

a) ¿Cuál es el *output* del algoritmo?

Una lista con los n números enteros de a, pero ordenados de menor a mayor.

b) ¿Cuál es la complejidad del algoritmo en función de n y k?

El primer **for** i es $O(k)$; el **for** j es $O(n)$; y el siguiente **for** i es $O(n)$, ya que esencialmente imprime los n datos de a, una vez cada uno. **Luego, el algoritmo es $O(k + n)$**.

c) Si k es $O(n)$, ¿cuál es la complejidad del algoritmo?

$O(n)$.

d) Esto pareciera contradecir la demostración que vimos en clase de que los algoritmos de ordenación por comparación tienen complejidad $O(n \log n)$. Explica por qué no hay una contradicción.

*Efectivamente, es un algoritmo de ordenación, **pero no ordena por comparación**: en ninguna parte del algoritmo aparece una condición tal como **a**[i] < **a**[j]. (O tal vez pdría argumentarse que la asigna-ción count[a[j]] = count[a[j]] + 1 hace una comparación de un valor contra otros n–1 en tiempo O(1).)*

2018-1 I1

3. Ordenación por comparación de elementos adyacentes

Sea *a* un arreglo de números distintos. Si $j < k$ y $a[j] > a[k]$, entonces el par (j, k) se llama una *inversión* de *a*.

a) ¿Cuál es exactamente el número promedio de inversiones que puede tener un arreglo *a* de *n* números distintos? (Hint: Considera el arreglo *a* y el arreglo *a* totalmente invertido, que llamamos *a'* ; entonces el par (j, k) es una inversión de *a* o es una inversión de *a'*.) Justifica.

Respuesta

Dado cualquier arreglo *a* y su inverso *a'* el pal de elementos (i, j) es una inversión en alguno de los dos arreglos. En total hay n*(n-1)/2 pares de elementos, por lo que en promedio hay n*(n-1)/4 inverisones.

b) A partir de (a), justifica que cualquier algoritmo de ordenación que ordena intercambiando elemen-tos adyacentes —por ejemplo, *insertionSort*— requiere tiempo $\Omega(n^2)$ en promedio para ordenar *n* ele-mentos.

Respuesta

Al intercambiar dos elementos adyacentes, sólo resolvemos una inversión a la vez. Como en promedio hay $n(n-1)/4$ inversiones, el algoritmo necesita realizar $n(n-1)/4 = \Omega(n^2)$ intercambios en promedio.

c) ¿Cuál es la relación entre el tiempo de ejecución de *insertionSort* y el número de inversiones del arreglo de entrada? Justifica.

Respuesta

Dado un arreglo *a* de entrada de *n* elementos, siempre debe iterar sobre el arreglo completo, por lo que como mínimo toma tiempo $\Omega(n)$. Luego, por cada inversión insertion sort hace un intercambio. Por lo tanto, el tiempo de insertion sort es O(n + inversiones), lo que en el mejor caso es O(n) y en peor caso es $O(n^2)$.

4. mergeSort y quickSort

b) Como vimos en clase, el desempeño de *quickSort* depende fuertemente de cómo resultan las particiones, lo que a su vez depende de cuál elemento del (sub)arreglo se elige como pivote.

- Explica cuál es la relación entre el resultado de las particiones y el desempeño de *quickSort*.
- Explica los pro y los contra de elegir como pivote el elemento de más a la derecha del (sub)arreglo, como es el caso de la versión del algoritmo que vimos en clase.
- Otro método para elegir el pivote es el llamado "la mediana de 3": se elige como pivote la mediana entre el elemento de más a la izquierda del (sub)arreglo, el elemento al medio del (sub)arreglo y el elemento de más a la derecha del (sub)arreglo. ¿Qué ventajas presenta este método frente al anterior?

Respuesta

Particiones vs desempeño: Es importante la forma en que se obtiene la partición en cada llamado a _quicksort_ pues si esta es muy desbalanceada, habrá un impacto sobre la complejidad del algoritmo completo. En efecto, lo ideal es que las partes obtenidas luego de particionar un subarreglo sean de tamaños parecidos para que se realicen O(nlog(n)) llamados en total. Para lograr este objetivo, no basta fijar una posición del pivote para particionar, sino que además se debe considerar si el arreglo ya está ordenado/semi-ordenado antes de comenzar.

Pivote extremo-derecho: Si se toma siempre el elemento extremo-derecho de cada sub-arreglo el algoritmo toma O(n^2) cuando el arreglo original está casi ordenado (respectivamente, ordenado de mayor a menor). Esto ocurre debido a que al armar las particiones, muy pocos elementos serán mayores (resp. menores) que el pivote y por lo tanto, el llamado recursivo siguiente se hará para un sub-arreglo de tamaño casi igual al actual. El peor caso es cuando el arreglo ya está ordenado, en cuyo caso cada llamado se hará para un sub-arreglo con un elemento menos que el paso anterior.

Mediana de 3: La ventaja inmediata es que este método no cae en los peores casos de usar como pivote el extremo-derecho o el extremo-izquierdo. Si el arreglo está ordenado/ordenado de mayor a menor, como el elemento central es la mediana de los tres elementos escogidos, no se hará un llamado con un sub-arreglo de largo similar al actual. En cualquier otro caso, el método no presenta mayor ventaja que usar un pivote aleatorio.

NOTA: Existe pregunta que considera el algoritmo de merge sort.

2018-1 EXAMEN
6. Ordenación y estadísticas de orden

Se quiere hacer un estudio de salud sobre la población chilena. Para esto, se registraron varias métri-cas sobre N personas elegidas aleatoriamente; una de estas métricas es la estatura de las personas. Para eliminar valores muy extremos (*outliers*), se decidió considerar solo las estaturas desde la i-ésima persona más alta hasta la j-ésima persona más alta; lamentablemente, los valores de las estaturas están desordenados.

Dados un arreglo *datos* con las N estaturas y los valores i y j, escribe un algoritmo en pseudocódigo que tenga tiempo esperado O(N) y que retorne las estaturas en el rango [i, j] (no importa si las retorna desordenadas).

P.ej., si *datos* = [1.80, 1.65, 1.79, 1.56, 1.57, 1.70, 1.76, 1.66, 1.86, 1.92], $i = 3$, $j = 7$, entonces el output del algoritmo debe ser 1.65, 1.79, 1.70, 1.76, 1.66.

Respuesta:

Si utilizamos QuickSelect para encontrar el elemento en la posición i del arreglo vamos a tener todas las alturas mayores a la derecha y las menores a la izquierda. Dado esto podemos usar QuickSelect en el sub-arreglo datos[i+1:] y encontrar la posición j. Esto a su vez hace que los elementos a su izquierda sean menores a datos[j]. Por lo tanto, los elementos entre las posiciones i y j son los elementos buscados **[4ptos]**. La complejidad esperada del algoritmo es O(N) ya que QuickSelect toma tiempo esperado O(N) y estamos usando ese algoritmo 2 veces **[2ptos]**.

2017-2 I1

1. a) ¿Es correcto que si la rutina Partition siempre realiza el m´ınimo numero de intercambios posibles, se ´ garantiza que QuickSort ejecuta en el menor tiempo posible (es decir, esta en el mejor caso)? Argumente.

a) Falso (0.5 puntos). Los peores casos de QuickSort son aquellos en los que ocurren muchas particiones desbalanceadas. Una particion desbalanceada es cuando el pivote, luego de ´ Partition, queda al inicio o al final del arreglo, y esto no tiene relacion alguna con el n ´ umero de intercambios realizados (puede haber ´ una particion desbalanceada con muchos intercambios as ´ ´ı como una sin ningun intercambio). Si ocurren ´ muchas particiones desbalanceadas en QuickSort, entonces, cada llamada recursiva procesara un arre- ´ glo de $n - 1$ elementos (arreglo original menos el pivote), por lo que se llamara n-veces hasta llegar a un ´ arreglo de largo 1 (el Heap de llamadas se vuelve lineal) (1.5 puntos por argumentar). Otra forma ser´ıa mostrar un contraejemplo. Por ejemplo, ordenar un arreglo ordenado con QuickSort (2 puntos si el contraejemplo esta correcto y bien fundamentado).

NOTA: Existe pregunta que considera como respuesta algoritmos como couting sort, radix sort, bucket sort (pregunta 3).

2017-1 EXAMEN

1. Para cada una de las siguientes afirmaciones, diga si es verdadera o falsa, siempre justificando su respuesta.
a) Quick Sort es un algoritmo de ordenacion estable. ´ Respuesta: Falso. Basta con dar un contraejemplo.

3. Existen diversas maneras de implementar Quick Sort. Una de ellas es reemplazar *Partition* por una función *MedianPartition*, que es tal que *MedianPartition*(A, p, r) es el índice en donde se ubica la *mediana* del arreglo $A[p..r]$, una vez que éste está ordenado.

   a) Modifique el pseudo-código de QuickSort—mostrado abajo—de manera que use esta idea. Su pseudocódigo debe ser detallado y completo; es decir, no debe faltar ninguna función por implementar. Asegure, además, que el tiempo promedio de *MedianPartition* sea $O(n)$.

```
1  function Partition(A, p, r)
2      i ← p − 1
3      j ← p
4      while j ≤ r do
5          if A[j] ≤ A[r] then
6              i ← i + 1
7              A[i] ↔ A[j]
8          j ← j + 1
9      return i
10 procedure Quick-Sort(A, p, r)
11     if p < r then
12         q ← Partition(A, p, r)
13         Quick-Sort(A, p, q − 1)
14         Quick-Sort(A, q + 1, r)
```

   b) Argumente a favor de que el tiempo promedio de su función *MedianPartition* es $O(n)$.

   c) ¿Es posible alimentar a su versión de Quick Sort con un arreglo A para que tome tiempo $O(n^2)$? Argumente.

   d) ¿Cómo espera que su algoritmo funcione en la práctica? ¿Cree que una variante de esta idea podría funcionar mejor? ¿Cuál?

2015-2 EXAMEN

4. Un componente esencial del algoritmo de ordenación **quicksort()** es el algoritmo de partición. Considera el siguiente algoritmo **partition()**:

```
partition(a, p, r):
   x = a[r]
   i = p-1
   for j = p … r-1:
      if a[j] ≤ x:
         i = i+1
         exchange(a[i], a[j])
   exchange(a[i+1], a[r])
   return i+1
```

a) Muestra el funcionamiento de **partition()** en el arreglo **a** = [B, H, G, A, C, E, F, D].

Inicialmente, p = 0, r = 7. x = 'D', i = –1

Ejecutamos el ciclo for j, con j = 0, 1, …, 6:

j = 0 –> a[j] = 'B' < x = 'D' –> i = 0 e intercambia (los contenidos de) a[0] y a[0] –> a queda igual
j = 1 –> a[j] = 'H' > x = 'D' –> no pasa nada
j = 2 –> a[j] = 'G' > x = 'D' –> no pasa nada
j = 3 –> a[j] = 'A' < x = 'D' –> i = 1 e intercambia a[1] y a[3] –> a = ['B', 'A', 'G', 'H', 'C', 'E', 'F', 'D']
j = 4 –> a[j] = 'C' < x = 'D' –> i = 2 e intercambia a[2] y a[4] –> a = ['B', 'A', 'C', 'H', 'G', 'E', 'F', 'D']
j = 5 –> a[j] = 'E' > x = 'D' –> no pasa nada
j = 6 –> a[j] = 'F' > x = 'D' –> no pasa nada

Así, al salir del ciclo for j, i = 2, y a = ['B', 'A', 'C', 'H', 'G', 'E', 'F', 'D']

Por lo tanto, finalmente intercambia a[3] y a[7] –> **a = ['B', 'A', 'C', 'D', 'G', 'E', 'F', 'H'] y retorna i+1 = 3**

La explicación anterior es la muestra más detallada del funcionamiento de **partition()**. Al menos, hay que mostrar las líneas en las que el contenido de **a** cambia, y la línea final; es decir, las tres líneas que están en **bold**.

b) Demuestra que en todo momento durante la ejecución de **`partition()`**, un arreglo **`a`** cualquiera está dividido en cuatro sectores: **`a[p]`** … **`a[i]`**, que son valores menores o iguales que el pivote; **`a[i+1]`** … **`a[j-1]`**, que son valores mayores que el pivote; **`a[j]`** … **`a[r-1]`**, que son valores aún no procesados; y **`a[r]`**, que es el pivote.

Los que importan son los sectores **`a[p]`** … **`a[i]`** y **`a[i+1]`** … **`a[j-1]`**; simplemente mirando el código, los otros dos sectores son "obvios".

La afirmación es inicialmente verdadera, ya que los sectores **`a[p]`** … **`a[i]`** y **`a[i+1]`** … **`a[j-1]`** **no tienen elementos**, debido a los valores iniciales de los índices (los *rangos* de los índices son vacíos).

El procesamiento del arreglo **`a`** lo hace el ciclo **`for j`**, que revisa en orden cada uno de los valores **`a[p]`** … **`a[r-1]`**, lo que de paso demuestra que el sector **`a[j]`** … **`a[r-1]`** corresponde a los valores aún no procesados.

La revisión de **`a[j]`** produce un efecto sólo si **`a[j]`** ≤ **`x`** (el pivote **`x`** = **`a[r]`** es constante a lo largo de todo el ciclo):

- el efecto es que el sector **`a[p]`** … **`a[i]`**, de los elementos menores o iguales que el pivote, aumenta su tamaño en 1 (la instrucción **`i = i+1`**) para recibir a **`a[j]`** (por la vía de intercambiarlo con **`a[i]`**);
- en cambio, si **`a[j]`** > **`x`**, entonces lo único que ocurre es que **`j`** aumenta en 1.

Como el sector **`a[p]`** … **`a[i]`** es inicialmente vacío, lo anterior significa que sólo crece para recibir valores menores o iguales que el pivote, demostrando esta parte de la afimación.

Además, como cada valor ≤ **`x`** va a parar al sector **`a[p]`** … **`a[i]`**, y el sector **`a[j]`** … **`a[r-1]`** contiene los elementos aún no procesados, entonces el sector **`a[i+1]`** … **`a[j-1]`** necesariamente contiene valores ya procesados y mayores que **`x`**, demostrando esta otra parte de la afimación.

c) ¿Qué valor devuelve **`partition()`** cuando todos los elementos del arreglo **`a`** tienen el mismo valor?

Si todos los elementos de **`a`** son iguales, entonces para cada **`j`**, efectivamente **`a[j]`** ≤ **`x`**. Por lo tanto, para cada **`j`**, se incrementa **`i`** (y se intercambian elementos de **`a`**, pero esto no produce cambios en cómo se ve **`a`**); como **`j`** va de **`p`** a **`r-1`**, entonces **`i`**, que parte en **`p-1`**, llega a valer **`r-2`**.

Y como **`partition()`** siempre devuelve **`i+1`**, entonces **`partition()`** devuelve **`r-1`** cuando todos los elementos de **`a`** son iguales.