Universidad Católica San Pablo
Algoritmos y Estructura de Datos
Prof. Jorge Poco
Alumno: Moreno Vera Felipe Adrian

Práctica 3
Entrega: 14 de Julio, 2018

Linear Time Sorting

1 Sorting in place in linear time

Suppose that we have an array of n data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

(a) Give an algorithm that satisfies criteria 1 and 2 above.

Solución:

Estable: Los elementos mantienen el orden original.
In-Place: No usa memoria adicional.
Por lo cual, Counting Sort es un algoritmo de orden O(n+d), pero cuando el número de dígitos es constante, se vuelve O(n) y además es estable.

(b) Give an algorithm that satisfies criteria 1 and 3 above.

Solución:

Sabemos que quicksort, por una partición triple con elementos iguales (no todos, pero si en gran mayoría) puede tener hasta un orden de O(n). Pero en un QuickSort convencional, podríamos también escoger el menor de todos como pivote (o una cota inferior como pivote) con la partición Lomuto (que incrementa en bucle haciendo iteraciones), haciendo que todos los mayores se vayan a la derecha y todos los menores se vayan a la izquierda (que no hay) y de esta manera, el orden de ordenamiento es O(n) también.

(c) Give an algorithm that satisfies criteria 2 and 3 above.

Solución:

Un algoritmo que sea estable y además ordene *in-place* es el insertion Sort.
Basta con verificar la manera de ordenamiento que tiene basado en comparaciones.
Es *in-place* porque no requiere de memoria adicional.
Es estable debido a que sea A[i]=a[j] con $i < j$, A[i] se considera primero en orden.

Entonces A[0, ..., i] será un array ordenado que contenga a A[i] en su posición original y correcta.
Sea un $k \leq i$ tal que A[k] es A[i].
De tal forma que el elemento A[j] tiene que ser intercambiado en el array A[0, ..., j] ordenado contenga a A[k], A[j] no puede intercambiarse con A[k] debido a que son iguales.
Verificando su estabilidad.

(d) Can you use any of your sorting algorithms from parts (a)-(c) as the sorting method used in line 2 of `RADIX-SORT`, so that `RADIX-SORT` sorts n records with b-bit keys in $O(bn)$ time? Explain how or why not (see page 198 in the book).

Solución:

De la referencia del libro de Thomas H. Cormen, Third Edition, pág 198:

RADIX-SORT.A; d **for** *i = 1 to d* **do**
| use a stable sort to sort array A on digit i
end

Algorithm 1: RADIX SORT

De la parte (a), sabemos que counting sort es O(n) y para b-bits con valores entre 0 o 1, los podemos ordenar en tiempo O(b(n+2))=O(bn).

(e) Suppose that the n records have keys in the range from 1 to k. Show how to modify counting sort so that it sorts the records in place in $O(n + k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable? (Hint: How would you do it for $k = 3$?).

Solución:

Desde la referencia del Cormen, pág 195. Tenemos:

COUNTING-SORT.A; B; k
Inicializamos el vector C[0...k]
for *i = 0 to k* **do**
| C[i] = 0
end
for *i = 1 to length(A)* **do**
| C[A[i]] = C[A[i]] + 1
end
for *i = 1 to k* **do**
| C[i] = C[i] + C[i-1]
end
for *i = length(A) to 1* **do**
| B[C[A[i]]] = A[i]
| C[A[i]] = C[A[i]] - 1
end

Algorithm 2: COUNTING SORT

Donde A es el array original, B es el array de salida (por lo cual es no in-place) y k es el número de dígitos.
Para hacer un ordenamiento *in-place*, tendriamos que hacer un cambio en el algoritmo a partir del 3er for.

```
COUNTING-SORT-INPLACE.A; B; k
Inicializamos el vector C[0...k]
for i = 0 to k do
|  C[i] = 0
end
for i = 1 to length(A) do
|  C[A[i]] = C[A[i]] + 1
end
for i = 1 to k do
|  C[i] = C[i] + C[i-1]
end
/* Aseguramos el ordenamiento in-place al asignar B los elementos de C */
for i = 0 to k do
|  B[i] = C[i]
end
/* Empezamos un bucle que intercambie respetando las posiciones */
/* Definimos un iterador */
i=1
while i ≤ n do
|  /* Si A[i] esta in-place, pasa al siguiente */
|  if B[A[i] − 1] < i and i ≤ B[A[i]] then
|  |  i = i + 1
|  end
|  else
|  |  Swap(A[i], A[C[A[i]]])
|  |  C[A[i]] = C[A[i]] - 1
|  end
end
```

Algorithm 3: COUNTING SORT in-place

Vemos que al realizar intercambios, el algoritmo no es estable, ya que necesita de memoria adicional.
Para k = 3

2 Sorting variable-length items

(a) You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over all the integers in the array is n. Show how to sort the array in $O(n)$ time.

Solución:

Sea el grupo A_i que contiene i dígitos y denominados a un k_i a la cantidad de elementos del grupo A_i.
Como la cantidad total de dígitos es n, agrupamos por el número de dígitos en orden ascendente usando Counting Sort, esto ocurre en O(n).
Para cada agrupación usaríamos Radix Sort tomaría ordenar cada grupo con i dígitos:
$T(n) = \sum_{i=1}^{n}(k_i)i$
Esto genera:
$T(n) = O(n)$

(b) You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is n. Show how to sort the strings in $O(n)$ time. (Note that the desired order here is the standard alphabetical order; for example, $a < ab < b$.)

Solución:

Para el caso de strings, donde se cumple $a < ab < b$. Agrupamos los strings por su longuitud y los ordenamos, usaremos Radix Sort desde derecha a izquierda.
De manera similar al ejemplo de arriba, sea l_i la longuitud de cada string.Se ordenará hasta un máximo de $l_i + 1$ caracteres utilizando Counting Sort.
Para cada agrupación usaríamos Radix Sort tomaría ordenar cada grupo con i dígitos:
$T(n) = \sum_{i=1}^{n}(l_i + 1)$
Esto genera:
$T(n) = O(n)$

Randomized Algorithms

1 Probabilistic counting

With a b-bit counter, we can ordinarily only count up to $2^b - 1$. With R. Morris's **probabilistic counting**, we can count up to a much larger value at the expense of some loss of precision.
We let a counter value of i represent a count of n_i for $i = 0, 1, ..., 2^b - 1$, where the n_i form an increasing sequence of nonnegative values. We assume that the initial value of the counter is 0, representing a count of $n_0 = 0$. The `INCREMENT` operation works on a counter containing the value i in a probabilistic manner. If $i = 2^b - 1$, then the operation reports an overflow error. Otherwise, the `INCREMENT` operation increases the counter by 1 with probability $1/(n_{i+1} - n_i)$, and it leaves the counter unchanged with probability $1 - 1/(n_{i+1} - n_i)$.
If we select $n_i = i$ for all $i > 0$, then the counter is an ordinary one. More interesting situations arise if we select, say, $n_i = 2^{i-1}$ for $i > 0$ or $n_i = F_i$ (the ith Fibonacci number—see Section 3.2 in the book).
For this problem, assume that n_{2^b-1} is large enough that the probability of an overflow error is negligible.

(a) Show that the expected value represented by the counter after n `INCREMENT` operations have been performed is exactly n.

Solución:

Definimos variables aleatorias, X_i que representa al ith operación de INCREMENT.
Para determinar el valor deterinado por el contador después de n INCREMENTS (según el cálculo por bits) denominado V_n será la suma de todos los X_i.

$V_n = X_1 + x_2 + ... + x_n$

Para calcular el valor esperado, podemos aproximarlo mediante el cáculo de la esperanza.
$E[V_n] = E[\sum_{i=1}^{n} X_i]$
$E[V_n] = E[X_1] + E[X_2] + ... + E[X_n]$

Vamos a determinar el valor de cada $E[X_i]$.
Del enunciado, sabemos que incrementa en $(n_{i+1} - n_i)$ con una probabilidad de $1/(n_{i+1} - n_i)$.
Y en caso de no incrementar, o incrementar en 0, tendría probabilidad de $1 - 1/(n_{i+1} - n_i)$.
Entonces en un i-ésimo término, la esperanza de que incremente sería:
$E[X_i] = (0.Pr(Noincrementar)) + ((n_{i+1} - n_i).Pr(incrementar))$
$E[X_i] = (0.(1 - \frac{1}{(n_{i+1}-n_i)})) + ((n_{i+1} - n_i).(\frac{1}{(n_{i+1}-n_i)}))$
$E[X_i] = ((n_{i+1} - n_i).(\frac{1}{(n_{i+1}-n_i)}))$
$E[X_i] = 1$
Por lo cual, el valor de $E[V_n]$ sería:
$E[V_n] = E[X_1] + E[X_2] + ... + E[X_n] = 1 + 1 + ... + 1 = n$

Se ve que el valor esperado luego de n incrementos es n mismo.

(b) The analysis of the variance of the count represented by the counter depends on the sequence of the n_i. Let us consider a simple case: $n_i = 100i$ for all $i \geq 0$. Estimate the variance in the value represented by the register after n `INCREMENT` operations have been performed.

Solución:

Nos piden estimar el valor de la varianza de V_n, la cual se representa como:
$Var[V_n] = Var[X_1] + Var[X_2] + ... + Var[X_n]$

Como $n_i = 100i$, se tendrá la variación $(n_{i+1} - n_i) = 100(i+1) - 100i = 100$.
Luego, la probabilidad de incremento seria de $1/100$.

Entonces, del cáculo de la varianza tenemos:

$Var[X_i] = E[X_i^2] - E^2[X_i]$
Pero sabemos que $E[X_j^2] = (0^2.(1 - \frac{1}{(n_{i+1}-n_i)})) + ((n_{i+1} - n_i)^2.(\frac{1}{(n_{i+1}-n_i)}))$.
$Var[X_i] = (0^2.(1 - \frac{1}{(n_{i+1}-n_i)})) + ((n_{i+1} - n_i)^2.(\frac{1}{(n_{i+1}-n_i)})) - 1^2$
$Var[X_i] = (0.(\frac{99}{(100)})) + ((100)^2.(\frac{1}{(100)})) - 1$
$Var[X_i] = (100) - 1$
$Var[X_i] = 99$
Entonces, calculando la varianza:
$Var[V_n] = Var[X_1] + Var[X_2] + ... + Var[X_n] = 99 + 99 + ... + 99 = 99n$

Se ve que la varianza luego de n incrementos es 99n.

2 Quicksort with equal element values

The analysis of the expected running time of randomized quicksort in Section 7.4.2 assumes that all element values are distinct. In this problem, we examine what happens when they are not.

(a) Suppose that all element values are equal. What would be randomized quicksort's running time in this case?

Solución:

Si todos los elementos son iguales, la función PARTITION retornará q=r, donde q es el índice menor y r s el índice mayor del array A[q ... r], entonces todos los elementos A[p ... q-1] son iguales.
Por lo que la recurrencia sería:
$T(n) = T(n-1) + T(0) + \Theta(n)$
Esto equivale a decir:
$T(n) = \Theta(n^2)$

(b) The `PARTITION` procedure returns an index q such that each element of $A[p..q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1..r]$ is greater than $A[q]$. Modify the `PARTITION` procedure to produce a procedure `PARTITION'`(A, q, r), which permutes the elements of $A[p..r]$ and returns two indices q and t, where $p \leq q \leq t \leq r$, such that

- all elements of $A[q..t]$ are equal,
- each element of $A[p..q-1]$ is less than $A[q]$, and
- each element of $A[t+1..r]$ is greater than $A[q]$.

Like PARTITION, your PARTITION' procedure should take $\Theta(r-p)$ time.

Solución:

```
PARTITION'A; p, r
x = A[p]
i = h = p
for j = p+1 to r do
    if A[j] < x then
        y = A[j]
        A[j] = A[h + 1]
        A[h + 1] = A[i]
        A[i] = y
        i = i + 1
        h = h + 1
    end
    else if A[j] == x then
        Swap(A[h+1], A[j])
        h = h+1
    end
end
Return (i,h)
```

Algorithm 4: PARTITION'

(c) Modify the RANDOMIZED-QUICKSORT procedure to call PARTITION', and name the new procedure RANDOMIZED-QUICKSORT'. Then modify the QUICKSORT procedure to produce a procedure QUICKSORT'(p, r) that calls RANDOMIZED-PARTITION' and recurses only on partitions of elements not known to be equal to each other.

Solución:

Solo basta modificar la llamada a la función RANDOMIZED-PARTITION en la función principal de QUICKSORT:

```
QUICKSORT'A; p, r
if p < r then
    (q,s) = RANDOMIZED-PARTITION(A,p,r)
    QUICKSORT'(A,p,q-1)
    QUICKSORT'(A,t+1,r)
end
```

Algorithm 5: QUICKSORT' with RANDOMIZED PARTITION

Donde el array A[q ... s] tiene elementos iguales.

(d) Using QUICKSORT', how would you adjust the analysis in Section 7.4.2 to avoid the assumption that all elements are distinct?

Solución:

Poniendo los elementos iguales al pivote en la misma partición, nos ayudaria a que el pivote evite recurrir en el elementos iguales. Por lo que el tamaño de los array en QUICKSORT' cuando los elementos son iguales, son menores que el QUICKSORT convencional donde los elementos son distintos.

Graph Algorithms

1 Critical Edges

You are given a graph $G = (V, E)$ a weight function $w : E \to \Re$, and a source vertex s. Assume $w(e) \geq 0$ for all $e \in E$.
We say that an edge e is $u$pwards critical if by increasing $w(e)$ by any $\epsilon > 0$ we increase the shortest path distance from s to some vertex $v \in V$.
We say that an edge e is downwards critical if by decreasing $w(e)$ by any $\epsilon > 0$ we decrease the shortest path distance from s to some vertex $v \in V$ (however, by definition, if $w(e) = 0$ then e is not downwards critical, because we can't decrease its weight below 0).

1. Claim: an edge (u, v) is downwards critical if and only if there is a shortest path from s to v that ends at (u, v), and $w(u, v) > 0$. Prove the claim above.

   Solución:

   Debemos demostrar 2 cosas:

   I. Si (u,v) es *downwards critical* entonces existe un camino más corto entre s y v que termina en (u,v) con $w(u, v) > 0$

   Entonces por definiciòn, si (u,v) es *downwards critical* con $w(u, v) > 0$ y decrecemos el valor de w(u,v) en $\epsilon > 0$, el valor del camino más corto también decrecerá. Entonces el camino más corto s a v debe contener a (u,v).

   II. Si existe un camino más corto entre s y v que termina en (u,v) con $w(u, v) > 0$ entonces (u,v) es *downwards critical*

   Como el camino de s a v contiene y termina en el nodo (u,v) con $w(u, v) > 0$, si decrementamos el valor de con w(u,v) en $\epsilon > 0$, estaremos decrementando la longuitud del camino más corto en la arista (u,v). Por lo cual (u,v) es un *downwards critical*.

2. Make a claim similar to the one above, but for upwards critical edges, and prove it.

   Solución:

   Para que un nodo sea *upwards critical*, significa que dicho nodo debe estar contenido en el camino más pequeño aun así se incremente, es decir, que no debe haber otro camino menor porque sino el camino más corto cambiaria.

   Entonces, hagamos la afirmación a demostrar:
   Un nodo (u,v) es *upwards critical* si y sólo si existe un camino más corto entre s y v que termina en (u,v) con $w(u, v) > 0$ y además (u,v) es el único subcamino que une s-v.

Es decir, que todos los caminos más pequeños de s a v, terminan en (u,v).

Prueba:

I. Si (u,v) es *upwards critical* existe un camino más corto entre s y v que termina en (u,v) con $w(u,v) > 0$ y además (u,v) es el único subcamino que une s-v.

Entonces por definiciòn, si (u,v) es *upwards critical* con $w(u,v) > 0$ y incrementamos el valor de w(u,v) en $\epsilon > 0$, el valor del camino más corto, evalaurá si con el incremento de w(u,v) seguirá siendo el menor o quizá haya otro camino que sea el nuevo camino más corto.
Si existiera otro camino que une de s-v sin pasar por (u,v), entonces en algún momento, el camino más corto dejará de pasar por (u,v). Quitándole la propiedad de *upwards critical*, por lo que (u,v) deberá ser el único subcamino que une s-v.

II. Si existe un camino más corto entre s y v que termina en (u,v) con $w(u,v) > 0$ y además (u,v) es el único subcamino que une s-v entonces (u,v) es *upwards critical*

Como el camino de s a v contiene al nodo (u,v) con $w(u,v) > 0$, si incrementamos el valor de con w(u,v) en $\epsilon > 0$, estaremos incrementando la longuitud del camino de s a v que pasa por (u,v). Como (u,v) es el único subcamino entre s-v, entonces al incrementar w(u,v), estaríamos incrementando la longuitud del camino más corto. Por lo cual (u,v) es *upwards critical*.

3. Using the claims from the previous two parts, give an $O(E \log V)$ time algorithm that finds all downwards critical edges and all upwards critical edges in G.

   Solución:

   Usando las premisas anteriores, tenemos que el nodo (u,v) es el único nodo que une v con el resto de caminos hacia s.
   Utilizamos el algoritmo de Dijkstra con Fibonacci Heaps (debido a que este tiene es de orden $O(|E| + |V|log|V|)$.
   Pero para el problema, nos piden todos, entonces debemos hacer una iteración sobre los caminos de s-v para almacenar los *upwards critical* y *downwards critical*.
   Entonces, almacenamos todos los nodos *downwards critical* que cumplan d(u) + w(u,v) = d(v).
   Luego, todos los nodos *upwards critical* son los que tienen w(u,v) = 1 o más que conecten con v siendo mínimo.
   El tiempo de ejecución del algoritmo sería:
   Como todos los vértices llegan hasta v, entonces debe ser V=O(E). Por lo que Dijkstra es O(ElogV) del cual, almacenar los *downwards critical* toma O(E) porque hacemos O(1) en cada nodo final y almacenar los *upwards critical* toma O(V) en todos los vértices.
   Entonces el orden final es:
   O(ElogV +E +V) = O(ElogV)

2 True or False

Decide whether these statements are **True** or **False**. You must briefly justify all your answers to receive full credit.

1. If some edge weights are negative, the shortest paths from s can be obtained by adding a constant C to every edge weight, large enough to make all edge weights nonnegative, and running Dijkstra's algorithm.

   Solución:

   False, Sea C una constante de tal forma que las aristas:
   $e_1 \leq e_2 \leq ... \leq e_k \leq 0 \leq e_{k+1} \leq ... \leq e_n$, escogemos $C \geq 1 - e_1$. Tenemos:
   $e_1 + C \leq e_2 + C \leq ... \leq e_k + C \leq C \leq e_{k+1} + C \leq ... \leq e_n + C$

$0 \leq e_1 - e_1 \leq e_2 - e_1 \leq ... \leq e_k - e_1 \leq -e_1 \leq e_{k+1} - e_1 \leq ... \leq e_n - e_1$
Entonces, todas nuestras aristas son positivas, podemos aplicar Dijkstra.
Pero no garantiza encontrar el camino más corto.

Ejemplo Contradictorio:
Sean los nodos A, B y C de tal forma que:
d(A, B) = -2, d(B,C) = 3 y d(A,C) = 2.
Vemos que la distancia más corta es $d(A, B) + d(B, C) = 1 < d(A, C) = 2$.
Por lo que P seria el conjunto de caminos (A-B, B-C).
Si sumamos la constante C=3 (mínimo valor para asegurarnos que todos sean positivos) a cada término tendríamos:
d(A, B) = -2+C, d(B,C) = 3+C y d(A,C) = 2+C.
Haciendo:
$d(A, B) + d(B, C) = 1 + 2C > d(A, C) = 2 + C$.
$d(A, B) + d(B, C) = 7 > d(A, C) = 5$.
El nuevo camino más corto sería (A-C)que es diferente de P.
Que contradice el enunciado.

2. Let P be a shortest path from some vertex s to some other vertex t. If the weight of each edge in the graph is squared, P remains a shortest path from s to t.

   Solución:

   False, Actualmente los pesos mantienen un orden de longuitud y del más pequeño.
   Sea e = {e_1, e_2, ..., e_n} las aristas del grafo G(E,V).

   De tal forma que $e_1 \leq e_2 \leq ... \leq e_n$, donde un conjunto tomado de e esel *camino más pequeño* entre s y t denotado por P.
   Tal como el problema anterior, sean las aristas de P el camino más corto, pero al elevar al cuadrado, eliminamos los caminos negativos, haciendo una suma positiva de enteros, de donde sea K la suma de los cuadrados de las aristas de P.
   Existirá otro camino tal que la suma de cuadrados sea menor que K.

   Ejemplo Contradictorio:
   Sean los nodos A, B y C de tal forma que:
   d(A, B) = -2, d(B,C) = 3 y d(A,C) = 2.
   Vemos que la distancia más corta es $d(A, B) + d(B, C) = 1 < d(A, C) = 2$.
   Por lo que P seria el conjunto de caminos (A-B, B-C).
   Si elevamos al cuadrado cada término tendríamos:
   d(A, B) = 4, d(B,C) = 9 y d(A,C) = 4.
   Haciendo:
   $d(A, B) + d(B, C) = 13 > d(A, C) = 4$.
   El nuevo camino más corto sería (A-C)que es diferente de P.
   Que contradice el enunciado.

3. A *longest simple path* from s to t is defined to be a path from s to t that does not contain cycles, and has the largest possible weight.

   Given a directed graph G with nonnegative edge weights and two nodes s and t, the following algorithm can be used to either find a longest simple path from s to t, or determine that a cycle is reachable from s:

   - Negate all the edge weights.
   - Run Bellman-Ford on the new graph.
   - If Bellman-Ford finds a shortest path from s to t, return that as the longest simple path.

- Otherwise, declare that a cycle is reachable from s.

Assume t is reachable from s.

Solución:

True, Haciendo los nodos negativos obtenemos un nuevo grafo `G'`.
Aplicando Bellman-Ford, o bien obtenemos el pseudo camino más corto con pesos negativos del nuevo grafo `G'` o bien obtenemos un ciclo negativo.
Pero como todos los nodos fueron invertidos, el camino más corto del grafo negativo `G'` es el camino más largo del grafo original G, y los ciclos encontrados en el grafo negativo `G'` son los ciclos positivos encontrados en el grafo original G.