CS6033 Design and Analysis of Algorithms I
Homework Assignment 5 Part 2
11.07.2022

Group Members:
rk4305 (Sai Rajeev Koppuravuri)
sg7372 (Sriharsha Gaddipati)
vt2182 (Venu Vardhan Reddy Tekula)
vt2184 (Veeravenkata Raghavendra Naveenkumar Tata)

---

1. (5 points) What is the running time of QUICKSORT when the array A contains distinct elements and is sorted in decreasing order. Please provide justification of your answer.

   Ans:

   Each partition will always contain its smallest element as the pivot. This element is placed in the first subarray, where it is in its correct position. The second subarray contains the remaining elements. By this, the recurrence relation is:

   $$T(n) = T(n-1) + n.$$
   $$\implies \text{ Running time of } T(n) = \Theta(n^2).$$

2. (5 points) If the array contained many duplicate items, which would be a better partitioning algorithm: Hoare, or the one presented in class? (Please refer CLRS book for Hoare algorithm).

   Ans:

   Hoare's algorithm would perform better than Lomuto's algorithm, if the array has a lot of duplicate elements. This is due to the fact that Lomuto's algorithm results in more swaps as there are more duplicates.

3. (10 points) Using Figure 8.3 (Please refer CLRS book) as a model, illustrate the operation of RADIX- SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

   Ans:

   I $\rightarrow$ Initial array of given english words.

   A $\rightarrow$ Given array was sorted based on last character of every word.

   B $\rightarrow$ Sort the previous array of words based on last but one character of each word.

   C $\rightarrow$ Sort the previous array of words based on first character of each word.

COW		SE**A**		T**A**B		**B**AR
DOG		TE**A**		B**A**R		**B**IG
SEA		MO**B**		E**A**R		**B**OX
RUG		TA**B**		T**A**R		**C**OW
ROW		DO**G**		S**E**A		**D**IG
MOB		RU**G**		T**E**A		**D**OG
BOX		DI**G**		D**I**G		**E**AR
TAB		BI**G**		B**I**G		**F**OX
BAR	$\Longrightarrow$	BA**R**	$\Longrightarrow$	M**O**B	$\Longrightarrow$	**M**OB
EAR		EA**R**		D**O**G		**N**OW
TAR		TA**R**		C**O**W		**R**OW
DIG		CO**W**		R**O**W		**R**UG
BIG		RO**W**		N**O**W		**S**EA
TEA		NO**W**		B**O**X		**T**AB
NOW		BO**X**		F**O**X		**T**AR
FOX		FO**X**		R**U**G		**T**EA
I		**A**		**B**		**C**

4. (10 points) Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any comparison sort stable. How much additional time and space does your scheme entail?

   Ans:

   In insertion sort, the array is reorganized by inserting elements at their proper positions in the array. Merge sort divides the array into two equal halves and then they are combined in a sorted manner. Insertion and merge sorts are stable, but heapsort and quicksort are not.

   Quicksort is not stable; to make it stable we can preprocess the array by replacing each element with a pointer to its place in the array using pointers. To make any sorting algorithm stable, we can preprocess each element of an array by replacing it with an ordered pair (x, y); the first entry will be the value of the element and the second entry will be its index.

   For example, the array [6, 1, 5, 3, 4, 5, 5, 2] would become [(6, 1), (1, 2), (5, 3), (3, 4), (4, 5), (5, 6), (5, 7), (2, 8)]. We now interpret $(i, j) < (k, m)$ if $i < k$ or $i = k$ and $j < m$.

   The algorithm is guaranteed to be stable under this definition of less-than because each element in our new array will be distinct from those in the original array and the index comparison ensures that if a repeat element appeared later in the original array, it must appear later in the sorted array. This doubles the space required for this algorithm but does not change its running time asymptotically.

5. (20 points) 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:

   (a) The algorithm runs in O(n) time.

   (b) The algorithm is stable.

   (c) The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

      i. (3 points) Give an algorithm that satisfies criteria 1 and 2 above.

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

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

      iv. (3 points) 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.

      v. (8 points) 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?

Ans:

(a) The algorithm for sorting a set of records satisfying the above given constraints sorts in O(n) time and is stable. The COUNTING SORT algorithm displays the output result in the same order as they do in the input array.

(b) The QUICKSORT algorithm partitions its array into two parts in order to accomplish a quick sort. Partitioning is done by moving all elements $<= 0$ to one side of the partition while placing all elements > 0 on the other side. This process takes time O(n).

(c) INSERTION-SORT is a stable sorting algorithm. It can be used to sort in-place, making it ideal for situations where space is limited. To prove that Insertion sort is stable. We observe that A[i] = A[j] and i ¡ j. Since i ¡ j, A[i] will be considered first. The element A[i] will be shifted at its correct position by comparing the rest of the elements of the array A [1….i-1]. This will result in a sorted array A[1…..i] containing the original A[i] at some position. So, when we consider A[j], it cannot bypass A[k] because A[k]¡A[j]. Therefore, the original A[i] and original A[j] will preserve their relative order.

(d) Part (a) – Counting sort can be used.

RADIX-SORT is a method used to sort the numbers according to their place in a sequence. It moves from least significant digit to most significant digit. Yes, COUNTING SORT algorithm (a) is also used as a sorting method used in line 2 of RADIX-SORT. First sort the keys based on the least significant value using COUNTING SORT. Now sort the sorted array for each more significant digit using COUNTING SORT. Final output of this array will be a sorted array. Counting sort runs in O(n) times, and for b-bit keys with each bit value varies from 0 to 1, we can sort in O(b(n + 2)) = O(bn) time.

(e) The modified counting sort is not stable. The below modified algorithm runs with O(n+k) time

```
MODIFIED-COUNTING-SORT(A, k)
let C[0..k] be a new array
for i = 1 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
for i = 2 to k
    C[i] = C[i] + C[i - 1]
insert sentinel element NIL at the start of A
B = C[0..k - 1]
insert number 1 at the start of B
// B now contains the "endpoints" for C
for i = 2 to A.length
    while C[A[i]] != B[A[i]]
        key = A[i]
        exchange A[C[A[i]]] with A[i]
        while A[C[key]] == key
        //make sure that elements with the same
        //keys will not be swapped
            C[key] = C[key] - 1
remove the sentinel element
return A
```