Data Structure and Algorithm Analysis(H)
Southern University of Science and Technology
Mengxuan Wu
12212006

Work Sheet 7

Mengxuan Wu

Question 7.1

1.

After the first loop:

0	1	2	3
0	0	0	0

After the second loop:

0	1	2	3
2	3	1	1

After the third loop:

0	1	2	3
2	5	6	7

2.

1	2	3	4	5	6	7
				1		
			1	1		
			1	1		3
		1	1	1		3
	0	1	1	1		3
	0	1	1	1	2	3
0	0	1	1	1	2	3

Array B

0	1	2	3
2	4	6	7
2	3	6	7
2	3	6	6
2	2	6	6
1	2	6	6
1	2	5	6
0	2	5	6

Array C

Question 7.2

We proof this by loop invariant.

Loop invariant: At the beginning of each iteration of the for loop of lines 6-7, every element $C[j]$ in the subarray $C[0..i-1]$ counts how many elements in the original array are less or equal to j. And every element $C[l]$ in the subarray $C[i-1..k]$ counts how many elements in the original array are equal to l.

Initialization: Before the first iteration of the loop, $i = 1$, the subarrays are $C[0]$ and $C[1..k]$. Since the first two loops let $C[j]$ counts how many j are in the original array, and no element is smaller than 0, both subarrays are correct.

Maintenance: We know this equation holds for every element j in $0, 1, \cdots, k$:

$$\text{number of element } \leq j = \text{number of element } \leq (j-1) + \text{number of element } = j$$

Since $C[i-1]$ counts how many elements in the original array are less or equal to $i-1$, and $C[i]$ counts how many elements in the original array are equal to i, we know that $c[i] + c[i-1]$ counts how many elements in the original array are less or equal to i. So we let $C[i] = C[i] + C[i-1]$ and the invariant holds.

Termination: When the loop terminates, $i = k+1$, the first subarray is $C[0..k]$ and the second is empty. Now we know that every element $C[j]$ counts how many elements in the original array are less or equal to j.

Question 7.3

3.

The algorithm will still sort the array correctly, but every element that has the same value will be in reversed order.

This is because after the third loop, every element $C[j]$ represents where the last j is in the sorted array. And $C[j]$ will be updated to $C[j] - 1$ once a j is put into the sorted array. This means no matter how we read the original array, we will always put the first j we read into the last position of all sorted j.

If we read the original array from the beginning, we will put the first j we read into the last position of all sorted j. The second j we read will be put into the second last position of all sorted j. And so on.

Therefore, the algorithm will still sort the array correctly, but every element that has the same value will be in reversed order.

Question 7.4

```
CountingRange(A, k, a, b)
1.  let C[0..k] be a new array
2.  n = A.length
3.  for i = 0 to k do
4.      C[i] = 0
5.  for j = 1 to n do
6.      C[A[j]] = C[A[j]] + 1
7.  for i = 1 to k do
8.      C[i] = C[i] + C[i − 1]
9.  if a = 0 then
10.     return C[b]
11. else
12.     return C[b] − C[a − 1]
```

Question 7.5

Original		After first loop		After second loop		After third loop
COW		MOB		TAB		BAR
DOG		TAB		BAR		BIG
TUG		DOG		CAR		BOX
ROW		TUG		TAR		CAR
MOB		PIG		PIG		COW
BOX		BIG		BIG		DOG
TAB	→	BAR	→	MOB	→	MOB
BAR		CAR		DOG		PIG
CAR		TAR		COW		ROW
TAR		COW		ROW		TAB
PIG		ROW		WOW		TAR
BIG		WOW		BOX		TUG
WOW		BOX		TUG		WOW

Question 7.6

Stable: InsertionSort, MergeSort
Not stable: HeapSort, QuickSort

For InsertionSort, if we read the original array from the beginning ,and when we insert an element into the sorted array, we always stop at the first position of all elements that have the same value. This means that the relative order of elements that have the same value will not change.

For MergeSort, when we merge two sorted arrays, we always put the element from the first array first. This means that the relative order of elements that have the same value will not change.

For HeapSort, it is not stable. For example, if we have an array $A = [1_a, 1_b, 1_c]$, which is already a max-heap. The sorted array will be $A = [1_a, 1_c, 1_b]$, because we take the root

of the heap first.

For QuickSort, it is not stable. For example, if we have an array $A = [2_a, 2_b, 1]$. The sorted array will be $A = [1, 2_b, 2_a]$. Because at the end of the first partition, 1 and 2_a will be swapped.