

COT 5405 Homework 1 Solutions

February 4, 2009

1 Grading Policy

- Maximum Score is 100.
- Each completed question is worth 10 points.
- Question 5 is graded for additional 50 points. Each subpart is worth 10 points.
- For this time only, no points cut for exceeding the page requirement which is one sheet per question. Please note that, next time it will have a penalty.
- For this time only we accepted late submissions and penalized to 10 points reduction. Please note that, next time no late submissions will be accepted.
- If you have a similar answer to an answer from somewhere else or someone and if you did not give any citations/references/people who discussed related to your answer, you lost additional 10 points. Note that, you can discuss your answers or explore resources, but you need to give references to your sources. This penalty will be increased for the upcoming homeworks, in case it happens again.
- Please read the regrading policy from the course syllabus. Contact TA Hale Erten regarding issues about grading (via email or during her office hours).
- Please let us know, if you find any mistakes in this solution.

2 Problem 2-4

(a) Inversions of the array $(2, 3, 8, 6, 1)$: $(2, 1)$, $(3, 1)$, $(8, 1)$, $(6, 1)$, $(8, 6)$ or indices $(1, 5)$, $(2, 5)$, $(3, 5)$, $(4, 5)$, $(3, 4)$

(b) The reverse order sorted array $n, n - 1, \dots, 2, 1$ will have the most inversions. Note that, there will be $n - 1$ inversions with the element 1, $n - 2$ inversions with the element 2, ..., and 1 inversion with the element $n - 1$. So, the total

number of inversions will be $\sum_{i=0}^{n-1} i = (n-1) * n/2$.

(c) The running time of insertion sort is proportional to the number of inversions in A, since the number of inversions is equal to the number of comparisons done as well as the number of swaps done executing insertion sort on array A. Consider the following argument: Let A be the input array (a_1, a_2, \dots, a_n) . Suppose we are at the i th iteration of the for loop, this means the first $i-1$ elements are in sorted order and $key = a_i$. Assume that (p, q) be an inversion, such that $a_p > a_q$ and $p < q$. In general, (p, i) form inversions will have $p < i$. Suppose there are $i-k$ such element locations p . If (p, i) was an inversion before running insertion sort on the subarray $A[1], \dots, A[i-1]$, then (p, i) is still an inversion of the resulting array, because the first $i-1$ loops only modify the position of the first $i-1$ elements. The i th loop will insert $key = a_i$ into the sorted subarray $A[1], \dots, A[i]$ at position k so that $a_1 \leq \dots \leq a_k = key \leq \dots \leq a_i$, for $1 \leq k \leq i$. The key will be compared to $(i-1) - (k-1) = i-k$ elements. All elements a_k, \dots, a_{i-1} will be shifted one position. Therefore, the number of elements that change position in the array when inserting element i is $(i-1) - k + 1 = i-k$ (we add 1 because the key is also moved). We assumed the number of inversions for element i was $i-k$ and showed that the number of comparisons made and the number of elements that moved to insert element i was $i-k$.

(d) The idea is to modify the combining procedure of merge sort (line 12-17 p. 29). Each time we combine two arrays we add to the total inversion count the number of inversions that have been fixed when combining the two subarrays. When an element from the right array is added to A, add to the inversion count the number of elements remaining in the left array. Add this number of inversions to the total number of inversions of A. Merge sort runs in time $\Theta(n \log n)$, and we have only increased the number of operations by a constant factor.

3 Problem 3-2

A	B	O	o	Ω	ω	Θ
$lg^k n$	n^ϵ	yes	yes	no	no	no
n^k	c^n	yes	yes	no	no	no
\sqrt{n}	n^{sinn}	no	no	no	no	no
2^n	$2^{n/2}$	no	no	yes	yes	no
n^{lgc}	c^{lgn}	yes	no	yes	no	yes
$lg(n!)$	$lg(n^n)$	yes	no	yes	no	yes

4 Problem 4-7

(a) If the array is Monge then we have, $A[i, j] + A[k, l] < A[i, l] + A[k, j]$, for $1 \leq i \leq k \leq m$ and $1 \leq j \leq l \leq n$. Now putting $k = i + 1$ and $l = j + 1$ we get, $A[i, j] + A[i + 1, j + 1] \leq A[i, j + 1] + A[i + 1, j]$ for $1 \leq i \leq m - 1$ and $1 \leq j \leq n - 1$.

Now suppose $A[i, j] + A[k, l] < A[i, l] + A[k, j]$, for $1 \leq i \leq k \leq m$ and $1 \leq j \leq l \leq n$ holds. We need to show that the array is Monge. We prove this by induction.

Basis: For any 2×2 subarray of the given array, the subarray is always a Monge array.

Induction: We first show that any $2 \times n$ subarray is also a Monge array. Assuming $2 \times (n - 1)$ subarray is a Monge array. So, the following holds,

$$A[1, 1] + A[2, n - 1] \leq A[2, 1] + A[1, n - 1]$$

$$A[1, n - 1] + A[2, n] \leq A[2, n - 1] + A[1, n]$$

Therefore, $A[1, 1] + A[2, n] \leq A[2, 1] + A[1, n]$ also holds. So, $2 \times n$ subarray is a Monge array. Now, by similar arguments and induction we can show that any $m \times n$ array is a Monge array if the given condition holds.

(b) Change $A[1, 3]$ from 24 to 22. Note that there are multiple answers for this problem.

(c) We can show this by contradiction, suppose $f(1) > f(2)$. Now, $A[1, f(1)]$ is the left most minimum value in row 1. So, $A[1, j] > A[1, f(1)]$ for all j from 1 to $f(1) - 1$. A is a Monge array. So, $A[1, f(2)] + A[2, f(1)] \leq A[1, f(1)] + A[2, f(2)]$ assuming $f(2) < f(1)$. But $A[2, f(2)]$ and $A[1, f(1)]$ are minimum values in their rows. So, $A[1, f(2)] + A[2, f(1)] > A[1, f(1)] + A[2, f(2)]$. This is a contradiction. Similar arguments apply for all $f(i)$ and $f(i + 1)$.

(d) From part c we know that $f(1) \leq f(2) \leq \dots \leq f(m)$. To find $f(2i + 1)$ such that $2i + 1 < m$, we must find minimum of $f(2i + 2) - f(2i) + 1$ values. For, $m/2$ rows (assuming m is even, without loss of generality) $f(2) - 0 + 1 + f(4) - f(2) + 1 + \dots + f(m - 2) - f(m - 4) + 1 + f(m) - f(m - 2) + 1$. This is same as $f(m) + m/2$. But we know $f(m) = O(n)$. So the cost is $O(m + n)$.

(e) At each recursive call the number of rows becomes half. So the recurrence is

$$T(m, n) = T(m/2, n) + O(m + n)$$

$$T(m, n) = [T(m/4, n) + O(m/2 + n)] + O(m + n)$$

...

$$T(m, n) = (T(m/2k, n) + O(m/2k - 1, n) + \dots + O(m + n))$$

Here $k = \lg m$. And $m + m/2 + \dots + m/2^{\lg m} = O(m)$. Therefore $T(m, n) = O(m + n \lg m)$.

5 Problem 7-4

(a) QUICKSORT' does the same as QUICKSORT algorithm does. It sorts the elements correctly. In first iteration of the while loop, it calls itself with $A, p, q - 1$, which is same as the first call in QUICKSORT algorithm. Then it sets the value of p to be $q + 1$ and goes in next iteration. Now, it performs the same function of second call in QUICKSORT.

(b) The stack depth is $\Theta(n)$. This happens when left partition has $n - 1$ elements and right partition has zero elements in all calls of QUICKSORT'.

(c) A $\Theta(\lg n)$ worst case stack depth can be achieved if we always pass the smallest of the partitions to first recursive call. Here is the code for it-

```

QUICKSORT"(A,p,r)
while  $p < r$  do
  do //partition and sort the small sub array first
   $q = \text{Partition}(A,p,r)$ 
  if  $q - p < r - q$  then
    QUICKSORT"(A,p,q-1)
     $p = q + 1$ ;
  else
    QUICKSORT"(A,q+1,r)
     $r = q - 1$ ;
  end if
end while

```

6 Problem 8-2

(a) Counting Sort can be used, since it is linear and stable. (Unless noted, counting sort is stable and not in-place due to additional B array).

(b) Partition algorithm of Quicksort. Considering the fact that we have n records and only two possible keys, 0/1, we can simply pass the array once with Partition to sort. Note that, Partition is not stable due to its swap operation.

(c) Insertion Sort is in-place and stable due to its shifting strategy during swapping, but its worst case running time is $O(n^2)$.

(d) Part (a) algorithm can be used since it is linear in time and stable which radix sort requires. Part (b) and (c) algorithms cannot be used, because (b) is not stable and (c) is not linear. Using (a) b times will take $O(bn)$ times as required.

(e) We need to eliminate the B array used in the Counting Sort algorithm, so that the algorithm does not require additional space in $O(n)$. Note that, we are allowed to use $O(k)$ amount of additional space. One strategy would be to count the numbers for each key value from 1 to k . Then, instead of going over array A , we can go over count array C and insert appropriate values to the array A positions. This part is not that straight forward, since we need to be careful not to overwrite the positions we already filled. A simple pseudocode can be given as follows:

```

COUNTING SORT'(A,k)
for  $i = 0$  to  $k$  do
  do  $C[i] = 0$ 
end for
for  $i = 0$  to  $\text{length}[A]$  do
  // count the number of elements for each key value
  do  $C[A[i]] = C[A[i]] + 1$ 
end for
 $count = 1$ 
for  $i = 0$  to  $k$  do

```

```

for  $j = C[i]$  downto 1 do
  do  $A[count] = i$ 
   $count++$ 
end for
end for

```

This algorithm is definitely not stable, but it still works in $O(n + k)$ time complexity. It is in-place, since we only use constant amount of additional space, $O(k)$.

Another approach would be to fill another array other than C of size k again. Like in counting sort, we will fill the C array to keep the location of the last occurrence of i in the sorted list. Let's have another array D to keep the first occurrence of the keys in the sorted list. This would be simply $C[i - 1] + 1$. Then, we will go over the array A and try to locate the element to the correct places.

COUNTING SORT'(A,k)

```

for  $i = 0$  to  $k$  do
  do  $C[i] = 0$ 
   $D[i] = 0$ 
end for
for  $i = 0$  to  $length[A]$  do
  // count the number of elements for each key value
  do  $C[A[i]] = C[A[i]] + 1$ 
end for
for  $i = 1$  to  $k$  do
  do  $C[i] = C[i] + C[i - 1]$ 
end for
for  $i = 1$  to  $k$  do
  do  $D[i] = C[i - 1] + 1$ 
end for
 $count = 0$ 
 $key = 1$ 
while  $count < length[A]$  do
  if  $D[A[key]] == key$  then
     $key = key + 1$ 
  else
    swap( $A[key], A[D[A[key]]]$ )
  end if
   $D[A[key]] = D[A[key]] + 1$ 
   $count = count + 1$ 
end while

```

This algorithm again takes $O(n + k)$ time while using $O(k)$ storage. Due to the swap operation, it is not stable.

Note: Solutions given are not the only possible solutions, there can be many different solutions.