Homework #3
Algorithms I
600.463
Spring 2017

Due on: Thursday, Feb 23rd, 5pm
Late submissions: will NOT be accepted
Format: Please start each problem on a new page.
Where to submit: On Gradescope under HW3.
Please type your answers; handwritten assignments will not be accepted.
To get full credit, your answers must be explained clearly,
with enough details and rigorous proofs.

March 9, 2017

1 Problem 1 (20 points)

Suppose an array $A[1 \dots n]$ of n elements, each of which is *red, white,* or *blue*. We want to sort the elements so that all the *reds* come before all the *whites*, which come before all the *blues*. The only permitted operations on the keys are:

- $Examine(A, i)$ — report the color of the ith element of A.
- $Swap(A, i, j)$ — swap the ith element of A with the jth element.

Design an efficient algorithm for the sorting. Prove the correctness and analyze the running time. A linear-time solution is expected.
Brief Answer:
Consider an algorithm as follows:
Color-Sorting$(A[], n)$:

```
1  l = 1, r = n
2  i = 1
3  while i <= r do
4      color=Examine(A, i)
5      if color == red then
6          Swap(A, i, l)
7          l+=1
8          i+=1
9      end
10     else if color == blue then
11         Swap(A, i, r)
12         r-=1
13     end
14     else
15         i+=1
16     end
17 end
```

The algorithm is quite straight-forward. First of all, lines 1-2 define three pointers: current under checking index i, red color index l (red items are less than index l), and blue color index r ($blue$ items are larger than r). The algorithm starts with checking i-th item until meets the index r: (1) If the item is red, swap to a place with index less than l. (2) If the item is $blue$, swap to a place with index larger than r.(3) otherwise, leave the $white$ item as is and check next item. In any iteration i of the while loop, the items that have been checked will be placed to three places:1 to $l-1$, l to i, or $r+1$ to n, according to their colors (for a more formal proof, this is your loop invariant). When i meets r, the array is sorted.

In total, since this algorithm iterates through the array A once and $Examine/Swap$ operation needs $O(1)$ time, the running time complexity is $O(n)$.

2 Problem 2 (20 points)

Assume that the array $A[1 \dots n]$ only has numbers from $\{1, \dots, n^{64}\}$ but that at most $\log \log n$ of these numbers ever appear. Design an algorithm that sorts A substantially less than $O(n \log n)$ time.

Brief Answer:

let's consider a simple "counting" algorithm as the following (assume the index starts at 1):

- Construct an empty 2D array $B[2][\log \log n]$ and initialize to all '0'. The first row $B[1][1 \ldots \log\log n]$ represents the distinct elements (keys) and the second row $B[2][1 \ldots \log\log n]$ represents the associated count of each distinct element (counts).

- Scan over the array A: for each element, do a linear scan on the first row of B, if the element is found in $B[1][i]$ for some i between 1 and $\log\log n$, increment the associated count in $B[2][i]$. If not found, insert the element into a '0' in the first row of B and update the count to 1.
- After all the elements in A are scanned and counted in B, sort the B based on the values of the first row. Here the counts in the second row of B are still associated with their keys.
- Copy the elements in B back to A in a sorted manner expect for '0' elements in B.

Since there are at most $\log\log n$ distinct elements, a 2D array $B[2][\log\log n]$ will be enough to count basically the "frequency vector" of the sequence A. The correctness follows from the correct linear scan and sorting algorithms. Here we can use any sorting algorithms to sort based on the values of B's first row, e.g. a slightly modified merge sort, quick sort, or even insertion sort.

Regarding the running time, for each element, it will take $\log\log n$ to perform a linear scan. So in total it needs $O(n \log\log n)$ since the sorting step needs $O(\log\log n \log(\log\log n))$, which is bounded by $O(n \log\log n)$.

Another valid solution could a $O(n)$ solution with large leading constant (e.g. 2*65). The idea is to define a radix sort with 65 digits and each digit can be 0 to $n-1$. Although $O(n)$ is better than $O(n \log\log n)$ in asymptotic analysis, $130 * n$ is much slower than $n \log\log n$ in practice.