CSCE 411-502 Design and Analysis of Algorithms

Spring 2025

Instructor: Dr. Jianer Chen
Office: PETR 428
Phone: (979) 845-4259
Email: chen@cse.tamu.edu
Office Hours: MW 1:30 pm–3:00 pm

Senior Grader: William Kang
Phone: 979) 575-9987
Email: rkdvlfah1018@tamu.edu
Questions: via phone and email and by appointments

Assignment #2 Solution

1. Suppose that in the algorithm CountingSort (see lecture notes on Feb. 10), we rewrite the **for**-loop header in step 3 as

```
for (i=0; i<n; i++),
```

i.e., we scan the array A[0..n-1] forwards. Modify the algorithm CountingSort properly so that the algorithm still sorts the array A[0..n-1] stably.

Hint: Recompute the array C[0..k-1] so that for each h, C[h] is the number of integers in A[0..n-1] that are *strictly smaller* than h.

Solution. As hinted above, we compute the array C[0..k-1] so that for each h, C[h] is the number of integers in A[0..n-1] that are strictly smaller than h.

```
CountingSort(A[0..n-1])
1.  for (h = 0; h < k; h++) C[h] = 0;
2.  for (i = 0; i < n; i++) C[A[i]] = C[A[i]] + 1;
\\ new C[h] = the number of h in the array A[0..n-1]
3.  prev = C[0]; C[0] = 0;
4.  for (i = 1; i < k; i++)
       temp = C[i];
       C[i] = prev + C[i-1];
       prev = temp;
\\ in loop-i:  C[i-1] = #values < i-1, prev = #values = i-1
5.  for (i = 0; i < n; i++)
       B[C[A[i]]] = A[i];
       C[A[i]] = C[A[i]] + 1;
```

Explanations:

(1) after steps 1-2, for each h, $C[h]$ is equal to the number of h's in A[0..n-1].

(2) For each i, $1 \leq i \leq k-1$, when the i-th execution of the for-loop in step 4 starts, we keep the following conditions: $C[i-1]$ is the number of values in A[0..n-1] that are strictly smaller than $i-1$, *prev* is the number of values in A[0..n-1] that are equal to $i-1$, and $C[i]$ is the number of values in A[0..n-1] that are equal to i. This

is certainly true for $i = 1$ because of step 3. Step 4 first saves the number of values that are equal to i in *temp*, then sets $C[i] = prev + C[i-1]$ so that $C[i]$ now becomes the number of values in A[0..n-1] that are either strictly smaller than $i-1$ or equal to $i-1$, i.e., $C[i]$ is the number of values in A[0..n-1] that are strictly smaller than i. The last line in step 4 assigns *prev* to *temp* so now *prev* becomse the number of values in A[0..n-1] that are equal to i. As a result, after the i-th execution of the for-loop in step 4, the values $C[i]$ and *prev* are ready for the $(i+1)$-st execution of the for-loop. **Remark:** note that implicitly here we have used induction on i to prove the correctness of step 4.

(3) By the analysis in (2), for each i, if $A[i] = h$, then $C[h] = C[A[i]]$ is exactly the first index for the first value h in the output (note that the array index starts from 0). Thus, the second line in step 5 places the value A[i] in the correct position in the output array B[0..n-1]. The third line in step 5 simply moves the index $C[h]$ to the next position for the next value h in the array A[0..n-1]. This also explains why this sorting algorithm is stable.

2. Assuming that you know that the elements of an array $A[n]$ are integers between 0 and $n^3 - 1$. Develop a linear-time algorithm that sorts $A[n]$.

Solution. Each integer x between 0 and n^3-1 can be written as $x = a_2n^2+a_1n+a_0$, where a_2, a_1, and a_0 are integers between 0 and $n-1$. Thus, if we treat each integer x between 0 and $n-1$ as a base-n 3-digit number $x = (a_2a_1a_0)_n$, then we can apply RadixSort to sort the array $A[n]$.

We discuss how we convert an integer x between 0 and $n^3 - 1$ into its base-n representation. Note that if we divide x by n, then the quotent is $x_1 = a_2n + a_1$ and the remainder is a_0. Now if we divide x_1 by n then the quotent is $x_2 = a_2$ and the remainder is a_1. The algorithm `Convert` below will do the conversion (using the operations in C++), and output the digit a_h, where h = 0, 1, 2, as given in the input:

```
int Convert(x, n, h)
1.  x1 = x / n;
2.  a_0 = x % n;
3.  a_1 = x1 % n;
4.  a_2 = x1 / n.
5.  output(a_h).
```

Thus, in constant time we can convert the integer x into its 3-digit base-n representation and retrieve any specific digit in the representation. Now we are ready to present the algorithm.

```
CountingSort(A[n],h)
\\ Sort array A[n] using its h-th digit in base-n representation
1.  for (h = 0; h < n; h++) C[h] = 0;
2.  for (i = 0; i < n; i++)
```

```
        k = Convert(A[i], n, h);
        C[k] = C[k] + 1;
3.   for (i = 0; i < n; i++) C[i] = C[i-1] + C[i];
4.   for (i = n-1; i >= 0; i--)
        k = Convert(A[i], n, h);
        C[k] = C[k] - 1;
        B[C[k]] = A[i].
5.   \\ copy the output back to the array A[n]
        for (i = 0; i < n; i++) A[i] = B[i];

main ()
   for (h = 0; h < 3; i++) CountingSort(A[n],h).
```

The `CountingSort` algorithm consists of 5 for-loops, each takes time $O(n)$. Thus, `CountingSort` takes time $O(n)$. The main algorithm calls `CountingSort` three times, so it also takes time $O(n)$.

3. Determine an LCS of $\langle 1,0,0,1,0,1,0,1\rangle$ and $\langle 0,1,0,1,1,0,1,1,0\rangle$.

Solution. The sequence $\langle 1,0,0,1,1,0\rangle$ is an LCS for the two given sequences, as indicated below.

$$\langle \underline{1}, \underline{0}, \underline{0}, \underline{1}, 0, \underline{1}, \underline{0}, 1\rangle$$
$$\langle 0, \underline{1}, \underline{0}, 1, 1, \underline{0}, \underline{1}, \underline{1}, \underline{0}\rangle$$

4. Develop an $O(nm)$-time algorithm that constructs the LCS for two sequences of lengths n and m, respectively, but uses only the array C[0..n, 0..m] without using the extra array B[0..n, 0..m].

Solution. The algorithm has been presented in class. We give some explanations here. As given in the slides of the lecture, when X[i] == Y[j], we will include the character X[i] in the LCS, while when X[i] $\neq$ Y[j], we will consider the LCS for the sequences X[1..i-1] and Y[j] and the LCS for the sequences X[1..i] and Y[j-1], and take the longer one. Thus, using the information given in X[i], Y[j], C[i-1, j], and C[i, j-1], we can completely determine how we should construct the LCS for X[1..n] and Y[1..m].

The algorithm for constructing the array C[0..n, 0..m] is the same as the one given in the lecture:

```
Dyn-LCS(X[1..n], Y[1..m])
1.   for (i=0; i<=n; i++) C[i,0] = 0;
2.   for (j=0; j<=m; j++) C[0,j] = 0;
3.   for (i=1; i<=n; i++)
```

```
        for (j=1; j<=m; j++)
           if (X[i]==Y[j])
              C[i,j] = C[i-1,j-1] + 1;
           else if (C[i-1,j] > C[i,j-1])
              C[i,j] = C[i-1,j];
           else C[i,j] = C[i,j-1].
```

Once the array C[0..n, 0..m] is contructed, we use the following algorithm to print the LCS.

```
Construct-LCS(C[0..n,0..m])
\\the length k of the LCS is C[n,m]
1.  i = n; j = m; k = C[n,m];
2.  while (i>0 & j>0)
       if (X[i]==Y[j])
          A[k]=X[i]; k = k - 1;
          i = i - 1; j = j - 1;
       else if (C[i-1,j] > C[i,j-1])
          i = i - 1;
       else j = j - 1.
3.  \\print the LCS
    for (i = 1; i < k; i++) output(A[i]).
```

Becuase the algorithms consist of simple for-loops, and because it is also easy to compute the running time of each execution of the for-loops, we can easily derive that the above algorithm runs in time $O(nm)$.