I111E Algorithms and Data Structures
Final Examination—Solutions

2019, Term 2-1

Ryuhei Uehara and Giovanni Viglietta (Room I67, {uehara,johnny}@jaist.ac.jp)

Problem 1: Let a and b be arrays containing n and m real numbers, respectively. We would like to count all the numbers that appear in both a and b. Analyze the computation time of each of the following three methods (here, you can assume that you can sort n numbers in $O(n \log n)$ time):

(1) For each element in a, check if it appears in b step by step.

(2) After sorting a, check if each element in b appears in a by binary search.

(3) After sorting a and b, check the common elements appearing in both of them through the merging process used in merge sort.

Solution:

(1) This algorithm takes the n elements of a and compares each of them with the m elements of b: in total, it performs nm comparisons, and its running time is therefore $O(nm)$.

(2) Sorting a takes $O(n \log n)$ time. Then, for each of the m elements of b, the algorithm performs a binary search in a, which takes $O(\log n)$ time. The total running time is therefore $O(n \log n) + O(m \log n) = O((n + m) \log n)$.

(3) Sorting a and b takes $O(n \log n)$ and $O(m \log m)$ time, respectively. The merging process used in merge sort takes $O(n + m)$ time. Hence, the total running time is

$$O(n \log n) + O(m \log m) + O(n + m) = O(n(\log n + 1) + m(\log m + 1) = O(n \log n + m \log m).$$

Problem 2: We are given a Binary Search Tree, and we want to print all the values stored in it in *increasing* order. Give an asymptotically optimal algorithm to perform this task. What is the running time of your algorithm?

Solution: Recall that the fundamental property of a BST is that, for each node, the left (respectively, right) subtree contains only values that are smaller (respectively, larger) than the value stored in the node itself. So, the idea is to start from the root and recursively print all the values in the left subtree, then the value stored in the current node, and finally all the values stored in the right subtree.

Here is a possible C implementation:

```
void BST_print(tree_node* root) {
    if (root == NULL) return;
    BST_print(root -> left);
    printf("%d ", root -> value);
    BST_print(root -> right);
}
```

This procedure visits each node exactly once, and in each node it performs a constant number of operations. Hence the running time is linear in the number of nodes, i.e., $O(n)$. Since the algorithm should at least print n values, and printing n values already takes $O(n)$ time, our solution is clearly optimal.

Problem 3: Let an array of n distinct integers $a[0] \sim a[n-1]$ be given. A *subsequence* is any subset of these numbers taken in order. An *increasing subsequence* is a subsequence where the numbers are getting increasingly large. Give an efficient algorithm that finds the longest increasing subsequence of the array.

(For example, the longest increasing subsequence of (4, 2, 7, 6, 3, 5, 8, 1, 9) is (2, 3, 5, 8, 9).)

Hint: Use dynamic programming to design an $O(n^2)$-time algorithm.

Solution: We define $L(i)$ as the subproblem of finding the length of the longest increasing subsequence of $a[\,]$ *terminating with* $a[i]$.

We store the values of $L(\cdot)$ in an array `length[]`. We construct `length[]` from left to right: to compute `length[i]`, we scan $a[\,]$ from $a[1]$ to $a[i-1]$, looking for a number $a[j]$ that is smaller than $a[i]$. When we find it, we know that we can use $a[i]$ to extend the longest increasing subsequence terminating with $a[j]$. We look up `length[j]` to see how long this subsequence is and, if it yields a subsequence of length greater than the value currently stored in `length[i]`, we update it. We also store the index j in `pred[i]`, so that later we can reconstruct the longest increasing subsequence terminating with $a[i]$. Throughout the entire process, we remember the largest value contained in `length[]` in a variable `max_length` and the corresponding index in `max_index`.

When we are finished constructing `length[]`, we have the length of the longest increasing subsequence in `max_length` and the index of its *last* element in `max_index`. To construct the longest increasing subsequence, we retrieve its elements in *reverse* order by looking up $a[\,]$ at index `max_index` and then going backwards following the indices stored in `pred[]`.

Here is a possible C implementation, which also prints the longest increasing subsequence after constructing it:

```
void long_incr_subs(int a[], int n) {
    int length[n];
    int pred[n];
    int max_length = 1;
    int max_index = 0;
    for (int i = 0; i < n; i++) {
        length[i] = 1;
        pred[i] = -1;
        for (int j = 0; j < i; j++) {
            if (a[i] > a[j] && length[j] + 1 > length[i]) {
                length[i] = length[j] + 1;
                pred[i] = j;
                if (length[i] > max_length) {
                    max_length = length[i];
                    max_index = i;
                }
            }
        }
    }
    int s[max_length];
    int index = max_index;
    for (int i = max_length - 1; i >= 0; i--) {
        s[i] = a[index];
        index = pred[index];
    }
    for (int i = 0; i < max_length; i++)
        printf("%d ", s[i]);
}
```

The running time is clearly quadratic, because the algorithm consists of two nested `for` loops that repeat for $O(n)$ steps each, followed by two consecutive `for` loops of length $O(n)$.

Problem 4: We are using the Fermat test to determine if $N = 65$ is prime.

(a) Is $a = 5$ a good or a bad witness? Why?

(b) Is $a = 64$ a good or a bad witness? Why?

(c) Is $a = 12$ a good or a bad witness? Why?

Hint: To compute a modular exponential by hand, you do not necessarily have to perform all the operations that a computer would, but you can use any "shortcut" that you come up with.

Solution: $N = 65$ is not prime, hence a is a *good* witness if $a^{64} \not\equiv 1 \pmod{65}$, and it is a *bad* witness if $a^{64} \equiv 1 \pmod{65}$.

(a) Observe that $\gcd(5, 65) = 5$. So, any positive power of 5 is still a multiple of 5 modulo 65. It follows that $5^{64} \not\equiv 1 \pmod{65}$, and $a = 5$ is a good witness.

(b) We have $64 \equiv -1 \pmod{65}$. So, $64^{64} \equiv (-1)^{64} \equiv ((-1)^2)^{32} \equiv 1^{32} \equiv 1 \pmod{65}$, and therefore $a = 64$ is a bad witness.

(c) We have $12^{64} \equiv ((12^2)^2)^{16} \equiv (144^2)^{16} \equiv (14^2)^{16} \equiv 196^{16} \equiv 1^{16} \equiv 1 \pmod{65}$, where we used the fact that $144 \equiv 14 \pmod{65}$ and $196 \equiv 1 \pmod{65}$. We conclude that $a = 12$ is a bad witness.

Problem 5: Suppose that the RSA cryptosystem did not use a composite modulus N, but simply a prime modulus p. Bob would have a public key (e, p), and Alice would encode her message x as $y = x^e \bmod p$. Bob would then retrieve x with his private key (d, p), by computing $x = y^d \bmod p$.

(a) How should Bob choose e and d so that the encoding and decoding phases work as intended?

(b) Show that this cryptosystem is not secure: that is, Eve can devise an algorithm that, given e, p, and $x^e \bmod p$ as input, efficiently computes x. Justify the correctness and analyze the running time of such an algorithm.

Solution:

(a) Bob should choose e and d so that they are inverses modulo $p-1$, i.e., $ed \equiv 1 \pmod{p-1}$. In practice, this is done by picking any e relatively prime to $p-1$, and then inverting it using the extended Euclid algorithm in $O(n^3)$ time. This choice works because, in the decoding phase, Bob computes $(x^e)^d \equiv x \pmod{p}$, and therefore he is able to retrieve Alice's original message x. Indeed, the above congruence holds because

$$x^{ed} \equiv x^{k(p-1)+1} \equiv (x^{p-1})^k \cdot x \equiv 1^k \cdot x \equiv x \pmod{p},$$

where we applied Fermat's little theorem: $x^{p-1} \equiv 1 \pmod{p}$.

(b) Given Bob's public key (e, p), Eve too can invert e modulo p by using the extended Euclid algorithm in $O(n^3)$ time. This way, Eve can compute Bob's private exponent d and use it to decode Alice's message (in the same way as Bob would) by modular exponentiation in $O(n^3)$ time. Since Eve is able to break the code in polynomial time, the cryptosystem is not secure.