I111E Algorithms and Data Structures

2019, Term 2-1

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

Problem 1 (5+5 points): We define the bit-wise operation $\oplus$ as follows: $0 \oplus 0 = 0$, $0 \oplus 1 = 1$, $1 \oplus 0 = 1$, and $1 \oplus 1 = 0$. If x and y are two non-negative integers, the operation $x \oplus y$ is defined by applying $\oplus$ to the corresponding digits in the binary representations of x and y.

For example, the result of $5 \oplus 14$ is 11, and it can be computed as follows:

$$\begin{array}{ccccc} & 0 & 1 & 0 & 1 \\ \oplus & 1 & 1 & 1 & 0 \\ \hline = & 1 & 0 & 1 & 1 \end{array}$$

Let x, y be two integer variables whose values are already set. We perform the following substitutions:

$x = x \oplus y;$

$y = x \oplus y;$

$x = x \oplus y;$

(a) Give some concrete values to x and y and check the sequence of substitutions.
(b) What do these substitutions aim at? (Justify your answer with a mathematical proof.)

Solution 1:

(a) Assume that $x = 5$ and $y = 14$.
After $x = x \oplus y$, we have $x = 5 \oplus 14 = 11$;
After $y = x \oplus y$, we have $y = 11 \oplus 14 = 5$;
After $x = x \oplus y$, we have $x = 11 \oplus 5 = 14$.
At the end, $x = 14$ and $y = 5$.

(b) These substitutions aim at exchanging the values of x and y. Les us prove it.
First observe that $\oplus$ enjoys four properties:

1. Identity: for every a, we have $a \oplus 0 = a$.
2. Nilpotence: for every a, we have $a \oplus a = 0$.
3. Commutativity: for every a and b, we have $a \oplus b = b \oplus a$.
4. Associativity: for every a, b, and c, we have $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.

To prove them, it is sufficient to show that they hold for numbers of a single binary digit.
Identity:

a	$a \oplus 0$
0	0
1	1

Nilpotence:

a	$a \oplus a$
0	0
1	0

Commutativity:

a	b	$a \oplus b$	$b \oplus a$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

Associativity:

a	b	c	$(a \oplus b) \oplus c$	$a \oplus (b \oplus c)$
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Now, let x_0 and y_0 be the initial values of x and y. Then,

$$x_1 = x_0 \oplus y_0,$$
$$y_1 = x_1 \oplus y_0,$$
$$x_2 = x_1 \oplus y_1.$$

We have to prove that $y_1 = x_0$ and $x_2 = y_0$.
By definition of x_1 and y_1, we have
$y_1 = x_1 \oplus y_0 = (x_0 \oplus y_0) \oplus y_0$.
By the properties of $\oplus$, we have
$y_1 = (x_0 \oplus y_0) \oplus y_0 = x_0 \oplus (y_0 \oplus y_0) = x_0 \oplus 0 = x_0$.
So, $y_1 = x_0$. We can substitute this equation in the definition of x_2:
$x_2 = x_1 \oplus y_1 = (x_0 \oplus y_0) \oplus y_1 = (x_0 \oplus y_0) \oplus x_0$.
By the properties of $\oplus$, we have
$x_2 = (x_0 \oplus y_0) \oplus x_0 = (y_0 \oplus x_0) \oplus x_0 = y_0 \oplus (x_0 \oplus x_0) = y_0 \oplus 0 = y_0$.
So, $x_2 = y_0$. We have proved that the final value of x is the initial value of y, and the final value of y is the initial value of x, and therefore the two variables have exchanged their values.

Problem 2 (5+5 points)

(a) Let $(a_n)_{n \geq 0}$ be the following sequence:

$$a_n = \begin{cases} n+1 & \text{if } n \in \{0,1,2,3\} \\ a_{n-1} + a_{n-4} & \text{otherwise} \end{cases}$$

Describe an efficient algorithm that takes n as input and outputs a_n.
What are the running time and space complexity of your algorithm?

(b) Let $(b_n)_{n \geq 0}$ be the following sequence:

$$b_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ b_{n-1} - b_{n-2} & \text{otherwise} \end{cases}$$

Describe an optimal algorithm that takes n as input and outputs b_n.
What are the running time and space complexity of your algorithm?

Solution 2

(a) This is structurally the same recurrence as the Fibonacci one, except that we have a_{n-4} instead of a_{n-2}. This means that we should remember the last 4 values of the sequence instead of the last 2. Here is a C program to compute this sequence efficiently:

```
int sequence_a(int n) {
    int last1 = 1;
    int last2 = 2;
    int last3 = 3;
    int last4 = 4;
    for (int i = 0; i < n; i++) {
        int next = last4 + last1;
        last1 = last2;
        last2 = last3;
        last3 = last4;
        last4 = next;
    }
    return last1;
}
```

The program runs in time $O(n)$ and uses $O(1)$ space.

(b) Let us compute the first terms of (b_n):

$$b_0 = 0,$$
$$b_1 = 1,$$
$$b_2 = 1,$$
$$b_3 = 0,$$
$$b_4 = -1,$$
$$b_5 = -1,$$
$$b_6 = 0,$$
$$b_7 = 1.$$

Since $b_6 = b_0$ and $b_7 = b_1$, the sequence repeats in the same way periodically, and the period is 6. So, to compute b_n, we just have to check n mod 6. Here is a C program to do that:

```
int sequence_b(int n) {
    if (n % 6 == 0) return 0;
    if (n % 6 == 1) return 1;
    if (n % 6 == 2) return 1;
    if (n % 6 == 3) return 0;
    if (n % 6 == 4) return -1;
    if (n % 6 == 5) return -1;
}
```

The program only performs a constant number of operations for every input. Hence, both the running time and the memory used are $O(1)$, which is obviously optimal.

Problem 3 (10 points) Let $s[\ ]$ be an array of size n with the following property: there exists an (unknown) index m such that the values $s[0]$, $s[1]$, $s[2]$, …, $s[m]$ are strictly increasing, and the values $s[m]$, $s[m+1]$, $s[m+2]$, …, $s[n-1]$ are strictly decreasing.

For example, if $s[\ ] = \{3, 8, 10, 11, 14, 12, 10, 9, 4, 2, 1\}$, then $m = 4$.

Describe an efficient algorithm that takes $s[\ ]$ (and n) as input and outputs m.
What is the running time of your algorithm?

Solution 3 Suppose we read two consecutive entries of the array: $s[i]$ and $s[i+1]$. By comparing them, we can determine if we have to move leftward or rightward to find m. Specifically, if $s[i] < s[i+1]$, then $m \geq i+1$; if $s[i] > s[i+1]$, then $m \leq i$.

We can use this observation to implement binary search on the array: at all times we have an interval in which we search for m, defined by its endpoints `left` and `right`. Then we examine the elements around the midpoint to determine if m lies in the left or in the right half of the interval, and we update `left` and `right` accordingly.

In doing so, we have to make sure that the indices of the array that we access are between 0 and $n-1$. That is, if the midpoint is 0, we skip the test to determine if m is to the left; if the midpoint is $n-1$, we skip the test to determine if m is to the right.

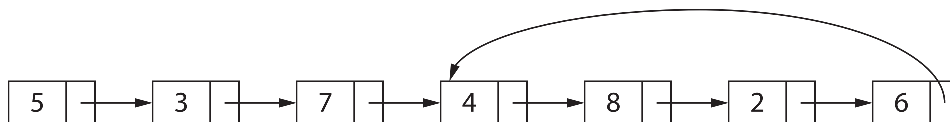
Here is a C program to perform the search:

```
int find_m(int s[], int n) {
   int left = 0;
   int right = n-1;
   do {
      int mid = (left + right) / 2;
      if ((mid < n-1) && (s[mid] < s[mid+1])) left = mid + 1;
      else if ((mid > 0) && (s[mid-1] > s[mid])) right = mid - 1;
      else return mid;
   } while (0 == 0);
}
```

Note that, by our assumptions on $s[\,]$, the index m that we are looking for always exists, and the algorithm always finds it. Therefore, it is unnecessary to put a terminating condition on the **`while`** loop: we chose to put `0 == 0` to make the program loop forever, i.e., until it returns m.

This algorithm has the same performance as binary search: the size of the interval is halved at every iteration of the loop; by the time the size becomes 1, m is found. At each iteration, a constant number of operations are executed. Therefore, the running time is $O(\log n)$.

Problem 4 (15 points) You are given a Linked List whose last node's pointer may be NULL or may point to some other node, thus forming a "loop", as shown in the figure:



Note that the sequential search algorithm performed on this Linked List may never terminate.

Describe a linear-time and constant-space algorithm that, given a Linked List as input, determines if it contains a loop or not (by printing "yes" or "no"). Note that the size of the Linked List is unknown. You may assume that the values stored in the nodes are all distinct.

Solution 4 The difficulty of the problem lies in the fact that, if the Linked List has a loop, we cannot scan it for a NULL pointer, because it contains none. So, determining when we have visited every node is not trivial (recall that we do not know how many nodes there are).

We could scan the Linked List and remember the value of every node we visit (for example using an array); once we reach a node containing a value that we have already seen, we conclude that the list has a loop. Unfortunately, this would be unacceptable, because it requires linear space.

The idea of our solution is to have two "runners" A and B scan the Linked List, where runner A is twice as fast as runner B. Runner B starts from the head of the Linked List, and runner A starts one step ahead. If the Linked List has no loop, then the two runners never meet, and runner A

eventually finds a NULL pointer: in this case, the algorithm terminates with a "no". If the Linked List has a loop, eventually both runners will be in the loop. Then, since one runner is faster than the other, eventually they will meet, and in this case the algorithm terminates with a "yes". To determine if the two runners have met, we simply compare the values stored in the nodes where they are currently located (recall that we assumed that all nodes have different values).
A possible C implementation is as follows:

```
void has_loop(list_node* head) {
   list_node* a = head -> next;
   list_node* b = head;
   int i = 0;
   while (a != NULL) {
      if (a -> value == b -> value) {
         printf("yes");
         return;
      }
      a = a -> next;
      if (i % 2 == 1) b = b -> next;
      i++;
   }
   printf("no");
}
```

Note that the pointer `a`, which represents the location of runner A, is updated at every iteration of the **while** loop. On the other hand, the pointer `b`, which represents the location of runner B, is updated only if `i` is odd, i.e., every two iterations. So, runner A moves twice as fast as runner B. Since we are using only three variables, the space complexity is constant.
To determine the running time, let n be the number of nodes in the Linked List. If there is no loop, runner A reaches the tail in $n-1$ iterations, and then the algorithm terminates: in this case, the time complexity is $O(n)$. If there is a loop, then the nth node of the Linked List points to, say, the mth node (and therefore the loop has length $n-m$ nodes). So, runner A enters the loop after m iterations, and runner B enters the loop after $2m$ iterations. After that, since runner A is twice as fast as runner B, their "distance" decreases by one node every two iterations. Initially, their distance within the loop is at most $n-m$ nodes, and so it becomes 0 after at most $2(n-m)$ iterations. We conclude that the two runners meet after at most $2m + 2(n-m) = 2n$ iterations, and therefore the algorithm runs in $O(n)$ time.

Problem 5 (10 points) Prove that the delete operation on a Binary Search Tree always preserves the BST property (i.e., for every node v, the left subtree contains only nodes that have values lower than v's, and the right subtree contains only nodes that have values higher than v's).

Solution 5 The delete operation has two cases: (1) the node v containing the value x has an empty child, and (2) the node v containing the value x has two non-empty children.

(1) Without loss of generality, suppose that the right child of v is empty. Let u be the parent of v, and let w be the left child of v. The delete operation removes v and puts w in its place, as a child of u. Suppose that v was the left child of u (the other case is symmetric): then, w becomes the left child of v, as well. This means that the values in the left subtree of u remain the same as before the deletion, except that now x is no longer among these values: since the BST property was satisfied for u before the deletion, it is satisfied now. Similarly, all the subtrees of all other nodes either remain unchanged after the deletion, or they lose v. The BST property is therefore satisfied for all nodes.

(2) In this case, we take the largest value y in the left subtree of v (let w be the node containing value y), and we put y into v instead of x. Then we delete w as in case (1) (we can do it, since

w has no right child, or else some other node in its right subtree would hold the maximum value). Let us prove that the combination of these two operations preserves the BST property for every node. Let u be the parent of w: the only two "critical" nodes are u and v. Indeed, the values in the subtrees of all other nodes either remain unchanged or they lose x: so, all these nodes keep satisfying the BST property. The proof for u is the same as in case (1), so the only node left is v. By assumption, all values in the original left subtree of v are smaller that the original value of v: in particular, $y < x$. Also, y is the largest of such values, and therefore the left subtree of v satisfies the BST property after the delete operation. As for the right subtree of v, its values never change, and they are all greater than x by assumption; since $y < x$, they are also greater than y. Therefore, the BST property is satisfied also by the right subtree of v after the delete operation.