Sample Solution for Problem Set 2

Data Structures and Algorithms, Fall 2019

September 27, 2019

10.1-6 (CLRS 3ed). We use Q to denote a queue, and $Q.S_1, Q.S_2$ to represent the two stacks implementing Q. We use the stack implementation shown on page 233 of CLRS 3ed. See Figure 1 for the detailed pseudocode.

```
1:  procedure ENQUEUE(Q, x)
2:      PUSH(Q.S_1, x)
3:  procedure DEQUEUE(Q)
4:      if STACK-EMPTY(Q.S_2) then
5:          while not STACK-EMPTY(Q.S_1) do
6:              x ← POP(Q.S_1)
7:              PUSH(Q.S_2, x)
8:      if STACK-EMPTY(Q.S_2) then
9:          return "overflow"
10:     else
11:         return POP(Q.S_2)
12: procedure QUEUE-EMPTY(Q)
13:     return STACK-EMPTY(Q.S_1) and STACK-EMPTY(Q.S_2)
```

Figure 1

The code works as follows: On each ENQUEUE operation, we just push the element into stack $Q.S_1$. On each DEQUEUE operation, we try to pop from $Q.S_2$. In case $Q.S_2$ is empty, we move all elements from $Q.S_1$ to $Q.S_2$ using stack operations.

The basic idea of the implementation is: At any time, the head of Q is the top of $Q.S_1$, and the queue moves towards the bottom of $Q.S_1$. Then, the next item in the queue is the bottom of $Q.S_2$, and the queue moves towards the top of $Q.S_2$. Finally, the tail of the queue is the top element in $Q.S_2$.

Regarding time complexity, it is easy to see ENQUEUE and QUEUE-EMPTY each takes $O(1)$ time. Moreover, within a series of n operations, each DEQUEUE operation takes $O(n)$ time. (However, this analysis is not tight. In particular, later in the course, we can use amortized analysis to show, the "average" time complexity for each operation is only $O(1)$.)

10.2-7 (CLRS 3ed). See Figure 2 for the pseudocode.

2.3-7 (CLRS 3ed). See Figure 3 for the pseudocode.

The code works in the following way. To being with, we sort the set S into ascending order. Then, we maintain two pointers l and r, such that l initially points to the first number in S and r initially points to the last number in S. The code then enters a loop with exit condition $l > r$. In the loop, whenever $S[l] + S[r] > x$, we decrease r by 1, and whenever $S[l] + S[r] < x$, we increase l by 1. If we can't find a pair (l, r) such that $S[l] + S[r] = x$ within the loop, we claim that no two numbers in S sum to x, and return false. Clearly, with a standard $O(n \log n)$ sorting algorithm, the total runtime of the FIND-SUM algorithm is $O(n \log n)$.

```
ReverseList(L)
1: new_head ← NIL
2: next ← NIL
3: cur_node ← L.head
4: while cur_node ≠ NIL do
5:     next ← cur_node.next
6:     cur_node.next ← new_head
7:     new_head ← cur_node
8:     cur_node ← next
9: L.head ← new_head
```

Figure 2

```
1: procedure FIND-SUM(S, x)
2:     SORT(S)
3:     l ← 1
4:     r ← S.length
5:     while l ≤ r do
6:         if S[l] + S[r] = x then
7:             return True
8:         else if S[l] + S[r] > x then
9:             r ← r − 1
10:        else
11:            l ← l + 1
12:    return False
```

Figure 3

We now argue the correctness of the algorithm.

If there do not exist two numbers in S that sum to x, clearly the algorithm will return false. So it remains to prove that if such pairs exist, our algorithm will successfully find one of them.

Suppose $S[l_0] + S[r_0] = x$, where $l_0 \leq r_0$. Our algorithm will eventually reach a state in which $l = l_0$ or $r = r_0$, before exiting. Without loss of generality, assume the algorithm reaches $l = l_0$ before $r = r_0$. Thus we have $r > r_0$. Since S is in sorted order, we have $S[l_0] + S[r] > S[l_0] + S[r_0] = x$. Therefore, r will decrease until it reaches r_0, at which point the algorithm correctly return "True".

2-4 (CLRS 3ed). **(a)** $(1, 5), (2, 5), (3, 4), (3, 5), (4, 5)$.

(b) $\langle n, n-1, \ldots, 2, 1\rangle$ has $\sum_{i=1}^{n-1} i = n(n-1)/2$ inversions.

(c) Essentially, every execution of line 6-7 "corrects" one inversion in the original input. Thus, the runtime of insertion sort is asymptotically equal the number of inversions in the original input.

(d) First of all, notice that when we merge two ordered arrays L and R, we can count the number of inversions in $\langle L, R\rangle$—which must have the form $L[i] > R[j]$—within $O(|L| + |R|)$ time.

As a result, similar to the merge sort algorithm, we can deploy the divide and conquer method to count the number of inversions in an array. Suppose array $A = \langle L, R\rangle$, then the number of inversions in A equals to the number of inversions in L, plus the number of inversions in R, plus the number of inversions of the form $L[i] > R[j]$.

So we can modify the merge sort algorithm to make it also return the number of inversions: For an input array $A = \langle L, R\rangle$, the number of inversions in L and R can be calculated recursively, and the number of inversions of the form $L[i] > R[j]$ can be calculated during the process of merging L and R. The sum of these three parts is the desired result. See Figure 4 for the complete pseudocode.

```
procedure CountInversions(A, p, r)
    if p < r then
        q ← ⌊(p + r)/2⌋
        l ← CountInversions(A, p, q)
        r ← CountInversions(A, q + 1, r)
        m ← Merge(A, p, q, r)
        return l + r + m
    else
        return 0
procedure Merge(A, p, q, r)
    cnt ← 0                                   ▷ Counter for number of inversions.
    n1 ← q − p + 1
    n2 ← r − q
    Let L[1..n1 + 1] and R[1..n2 + 1] be two new arrays
    for i = 1 to n1 do
        L[i] ← A[p + i − 1]
    for j = 1 to n2 do
        R[j] ← A[q + j]
    L[n1 + 1] ← ∞
    R[n2 + 1] ← ∞
    i ← 1
    j ← 1
    for k = p to r do
        if L[i] ≤ R[j] then
            A[k] ← L[i]
            i ← i + 1
        else
            A[k] ← R[j]
            cnt ← cnt + n1 − i + 1
            j ← j + 1
    return cnt
```

Figure 4

4.3-6 (CLRS 3ed). For $n \geq n_1 = 68$, it holds that $n/2 + 17 \leq 3n/4$. We'll find c and d such that $T(n) \leq cn \log n - d$.

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor + 17) + n \\ &\leq 2(c(n/2 + 17)\log(n/2 + 17) - d) + n \\ &\leq cn \log(n/2 + 17) + 34c \log(n/2 + 17) - 2d + n \\ &\leq cn \log(3n/4) + 34c \log(3n/4) - 2d + n \\ &= cn \log(n) - d + cn \log(3/4) + 34c \log(3n/4) - d + n \end{aligned}$$

Take $c = -2/\log(3/4)$ and $d = 34$, we have $T(n) \leq cn \log(n) - d + 34c \log(n) - n$. Since $\log(n) = o(n)$, there must exists n_2 such that $n \geq 34c \log(n)$ whenever $n \geq n_2$. Letting $n_0 = \max\{n_1, n_2\}$, we have that $T(n) \leq cn \log(n) - d$ whenever $n \geq n_0$. Therefore $T(n) = O(n \log n)$.

4.4-5 (CLRS 3ed). After drawing the first few levels of the recursion tree, you should have found that the cost for each level is increasing geometrically. Moreover, the height of the tree can reach n. Thus, we guess that the runtime is $O(2^n)$, and this indeed is true as

$$\begin{aligned} T(n) &= T(n-1) + T(n/2) + n \\ &\le 2^{n-1} + \sqrt{2^n} + n \\ &\le 2^n \end{aligned}$$

when n is sufficiently large.

This is actually a pretty good bound, as $T(n)$ does not have any polynomial upper bound. (Try to prove this.)

4.4-9 (CLRS 3ed). Without loss of generality, assume $\alpha \le 1 - \alpha$. Consider the recursion tree, we can see that the cost for each of the first $\lg_{1/(1-\alpha)} n$ layer is $\Theta(n)$, thus the total cost is at least

$$(\lg_{1/(1-\alpha)} n) \cdot \Theta(n) = \Theta(n \lg_{1/(1-\alpha)} n)$$

implying

$$T(n) = \Omega(n \lg_{1/(1-\alpha)} n) = \Omega(n \log n)$$

On the other hand, notice that the height of the tree is $\lg_{(1/\alpha)} n$, with each layer costing at most $O(n)$, thus we get the upper bound

$$T(n) = O(n \cdot \lg_{(1/\alpha)} n) = O(n \log n)$$

Combining these two bounds leads to the conclusion that $T(n) = \Theta(n \log n)$.

4.5-4 (CLRS 3ed). With $a = 4$ and $b = 2$, we have $f(n) = n^2 \cdot \lg n$. Since $f(n) \notin O(n^{2-\epsilon})$ and $f(n) \notin \Omega(n^{2-\epsilon})$, we cannot apply the master method.

We guess $T(n) \le cn^2 \lg^2 n$, thus

$$\begin{aligned} T(n) &\le 4T(n/2) + n^2 \lg n \\ &= 4c(n/2)^2 \lg^2(n/2) + n^2 \lg n \\ &= cn^2 \lg^2 n + (1-c)n^2 \lg n - cn^2 \lg(n/2) \end{aligned}$$

For $c \ge 1$, we have

$$\begin{aligned} T(n) &\le cn^2 \lg^2 n - cn^2 \lg(n/2) \\ &\le cn^2 \lg^2 n \end{aligned}$$

Thus $T(n) = O(n^2 \lg^2 n)$.

4-2 (CLRS 3ed). **(a.1)** $T(n) = T(n/2) + c$ where c is a constant. Apply the master method leads to $T(n) = \Theta(\lg n)$.

(a.2) $T(n) = T(n/2) + cN$, where N is the input array's length. Notice, N is a constant during the recurrent procedure. Using the recursion tree method we know $T(n) = \sum_{i=0}^{\lg n - 1}(2^i \cdot (cN/2^i)) = cN \lg n$. As a result, $T(N) = cN \lg N = \Theta(N \lg N)$.

(a.3) $T(n) = T(n/2) + cn$. Apply the master method leads to $T(n) = \Theta(n)$.

(b.1) $T(n) = 2T(n/2) + cn$. Apply the master method leads to $T(n) = \Theta(n \lg n)$.

(b.2) $T(n) = 2T(n/2) + cn + 2N$. Similar to our previous discussion, using the recursion tree method, we get $T(n) = \sum_{i=0}^{\lg n - 1}(cn + 2^i N) = cn \lg n + nN - N = \Theta(nN)$. Thus $T(N) = \Theta(N^2)$.

(b.3) $T(n) = 2T(n/2) + cn + 2n/2 = 2T(n/2) + (c+1)n$. Apply the master method leads to $T(n) = \Theta(n \lg n)$.

Additional Problem One. Use two arrays A and M, where A simulates the standard stack and $M[i]$ stores the maximum value in $A[0, \cdots, i]$.

Algorithm 1: MAX-STACK

```
cur ← 0;
function push(x):
    A[cur] ← x;
    if cur == 0 or x > M[cur − 1] then
        M[cur] ← x
    else
        M[cur] ← M[cur − 1]
    cur ← cur + 1
function pop():
    if cur == 0 then
        return NULL
    cur ← cur − 1;
    return A[cur]
function max():
    if cur == 0 then
        return NULL
    return M[cur − 1]
```

We need to prove M always satisfies the properties described above: $M[i]$ stores the maximum value among $A[0, \cdots, i]$, where $0 \leq i < cur$.

Prior to the first operation we have $cur = 0$, the claim is true.

Assume prior to a new operation the claim is true and $cur = cur'$, we need to show the claim still holds after this new operation.

If this new operation is $push$: After doing $push$, $M[0, \cdots, cur' - 1]$ and $A[0, \cdots, cur' - 1]$ remain unchanged, thus $M[i]$ stores the maximum value among $A[0, \cdots, i]$ for $0 \leq i < cur'$. Since $M[cur']$ is $\max(M[cur' - 1], A[cur'])$ and $A[cur'] = x$, we know $M[cur']$ stores the maximum value among $A[0, \cdots, cur']$. Thus, after the $push$ operation, the claim still holds.

If this new operation is pop: Prior to doing pop we have $M[i]$ stores the maximum value among $A[0, \cdots, i]$ for $0 \leq i < cur'$. Notice pop does not changes the values stored in A and M, thus after doing pop, $M[i]$ stores the maximum value among $A[0, \cdots, i]$ for $0 \leq i < cur' - 1$. This implies, after the pop operation, the claim still holds.

If this new operation is max: max operation does not change the value of cur, or the values stored in A and M. Trivially, the claim still holds after the max operation.

At this point, we have proved $M[i]$ stores the maximum value among $A[0, \cdots, i]$, where $0 \leq i < cur$. As a result, it is easy to see the max operation always returns the desired result.

The space complexity is $O(n)$ where n is the number of elements in the stack.

Additional Problem Two. **(a)** A simple divide-and-conquer algorithm is given in Algorithm 2. The runtime recurrence is $T(n) = 2T(n/2) + \Theta(n)$, implying $T(n) = \Theta(n \log n)$.

(b) You can use the procedure shown in Algorithm 3 as the hint suggests.

We now argue the correctness of the algorithm. To begin with, since we have done some pre-treatment when n is odd, we can assume n is even. If array A has a majority element k, let $K_1 = |\{i \in \{1, 3, 5, \cdots, n-1\} | A[i] = A[i+1] \; and \; A[i] = k\}|$, $K_2 = |\{i \in \{1, 3, 5, \cdots, n-1\} | A[i] = A[i+1] \; and \; A[i] \neq k\}|$, $K_3 = |\{i \in \{1, 3, 5, \cdots, n-1\} | A[i] \neq A[i+1] \; and \; (A[i] = k \; or \; A[i+1] = k)\}|$, $K_4 = n/2 - K_1 - K_2 - K_3$. Since k is a majority element, $2K_1 + K_3 > 2K_2 + K_3 + 2K_4$, implying $K_1 > K_2$. Notice, B contains K_1 copies of k and K_2 other values. Thus, k is again a majority element

Algorithm 2: FIND-MAJORITY-ELEMENT(A, l, r)

```
if l == r then
    return A[l]
m ← ⌈(l+r)/2⌉;
k1 ← FIND-MAJORITY-ELEMENT(A, l, m);
k2 ← FIND-MAJORITY-ELEMENT(A, m + 1, r);
cnt1 ← 0, cnt2 ← 0;
for i from l to r do
    if A[i] == k1 then
        cnt1 ← cnt1 + 1
    if A[i] == k2 then
        cnt2 ← cnt2 + 1
if cnt1 > ⌊(r−l+1)/2⌋ then
    return k1
if cnt2 > ⌊(r−l+1)/2⌋ then
    return k2
return −1
```

Algorithm 3: FAST-FME(A, n)

```
B ← NEW-ARRAY();
cnt ← 0;
if n == 0 then
    return −1
if n == 1 then
    return A[1]
if n%2 == 1 then
    for i from 1 to n do
        if A[i] == A[n] then
            cnt ← cnt + 1
    if cnt > ⌊n/2⌋ then
        return A[n]
    n ← n − 1
cnt ← 0;
for i from 1 to n step 2 do
    if A[i] == A[i + 1] then
        cnt ← cnt + 1;
        B[cnt] = A[i]
k ← FAST-FME(B, cnt);
cnt ← 0;
for i from 1 to n step 1 do
    if A[i] == k then
        cnt ← cnt + 1
if cnt > ⌊n/2⌋ then
    return k
else
    return −1
```

in B. As a result, we know if the input indeed has a majority element, then the algorithm will correctly find it. On the other hand, it is easy to see the algorithm will output -1 if there is no majority element.

The time complexity recurrence is $T(n) = T(n/2) + \Theta(n)$, implying $T(n) = \Theta(n)$.