Solution for Problem Set 2

20130005 方盛俊

Problem 1

(a)

Let L be the linked list.

Algorithm 1 Reversing Linked List

```
function REVERSED(L)
    head = L.getHead()
    tail = head
    while tail.next is not null do
        tail = tail->next
    end while
    for i = 1 to n do
        tail.next = &head
        tail = head
        head = head->next
        L.setHead(head)
    end for
    tail.next = null
end function
```

(b)

Overview:

We use `x.np` to simulate `x.next` and `x.prev`. Define `x.np = x.prev ⊕ x.next` so that we can get `x.prev` by `x.np ⊕ x.next` and get `x.next` by `x.np ⊕ x.prev`. We save a default node S named sentinel to simplify the operation. We define `S.np = &head ⊕ &tail`. And we need the list L provide (or save) S and the head of list.

Algorithm:

Let S be the **sentinel** and L.getHead() be the head of the link. If there are nothing in L, L.getHead() will be S itself.

Algorithm 2 Insert

```
function INSERT(x, i)
```

```
    last = S
    curr = L.getHead()
    isNewHead = (i == 1)
    if i > L.size() / 2 then
        curr = *(S.np ⊕ &head)
        i = L.size() - i + 2
    end if
    while i ≠ 1 do
        next = *(curr.np ⊕ &last)
        last = curr
        curr = next
    end while
    x.np = &last ⊕ &curr
    last.np = last.np ⊕ &curr ⊕ &x
    curr.np = curr.np ⊕ &last ⊕ &x
    if isNewHead then
        L.setHead(x)
    end if
    L.setSize(L.size() + 1)
end function
```

Algorithm 3 Delete

```
function DELETE(i)
    last = S
    curr = L.getHead()
    if i == 1 then
        L.setHead(*(curr.np ⊕ &last))
    end if
    if i > L.size() / 2 then
        curr = *(S.np ⊕ &head)
        i = L.size() - i + 2
    end if
    while i ≠ 1 do
        next = *(curr.np ⊕ &last)
        last = curr
        curr = next
    end while
    next = *(&last ⊕ curr.np)
    last.np = last.np ⊕ &curr ⊕ &next
    next.np = next.np ⊕ &curr ⊕ &last
    L.setSize(L.size() - 1)
end function
```

Problem 2

Overview:

We save the element x with the current maximum, which makes sure that the top of the stack always is the current maximum. And we still can get the old maximum after twice pop operation.

Algorithm:

Let S be the stack.

Algorithm 4 MaxStack

```
function MAX(x)
    if S.size() > 0 then
        max = S.pop()
        S.push(max)
        return max
    else
        return NULL
    end if
end function
function PUSH(x)
    if S.size() > 0 then
        max = Max()
        S.push(x)
        if x > max then
            S.push(x)
        else
            S.push(max)
        end if
    else
        S.push(x)
        S.push(x)
    end if
end function
function POP(x)
    if S.size() > 0 then
        S.pop()
        return S.pop()
    else
        return NULL
    end if
end function
```

Space Complexity:

We need to save the element and the current maximum for each pushed element, so the time complexity $S(n) = c_1n + c_2n = (c_1 + c_2)n = \Theta(n)$

Problem 3

Overview:

Let the input expression array be I, create a output array named O and two variances named `formerOp` and `latterOp`. Scan character in the input array I one by one. If current character `ch` is a number or operator $!$, we add it to the output array. If `ch == '+'`, we assign it to `formerOp` when `formerOp` is empty, or we replace with formerOp and add `'*'` to output array when `formerOp == '×'`, or we add `'+'` to output array when `formerOp == '+'`. If `ch == '×'`, we assign it to `formerOp` when `formerOp` is empty, or we assign it to `latterOp` when `formerOp == '+'` and `latterOp` is empty, or we add it to output array when `formerOp == '×'` or `latterOp == '×'`. Finally, we add `latterOp` and `formerOp` to output array.

Algorithm:

Let $I[1...n]$ be the origin infix expression.

Algorithm 5 Convert

```
function CONVERT()
    O[1...n]
    i = 1
    formerOp = null, latterOp = null
    for ch in I do
        if ch is number or ch == '!' then
            O[i] = ch
            i = i + 1
        else if ch == '+' then
            if formerOp == null then
                formerOp = '+'
            else if formerOp == '×' then
                formerOp = '+'
                O[i] = '×'
                i = i + 1
            else
                O[i] = '+'
                i = i + 1
            end if
        else if ch == '×' then
            if formerOp == null then
                formerOp = '×'
            else if formerOp == '+' and latterOp == null then
                latterOp = '×'
            else
                O[i] = '*'
                i = i + 1
            end if
        end if
    end for
    if latterOp != null then
```

$O[i] = \text{latterOp}$
$\text{i} = \text{i} + 1$
end if
if $\text{formerOp} \mathrel{!}= \text{null}$ **then**
$O[i] = \text{formerOp}$
end if
end function

Time Complexity:

The running time in the worst case:

$$T(n) = c_1 + c_2 n + c_3 = O(n)$$

Problem 4

Algorithm A:

- Let $T(n)$ be the runtime of A on instance of size n
- Clearly, $T(1) = c_1 = \theta(1)$ for some constant c_1
- $T(n) = 5 \cdot T(\frac{n}{2}) + c_2 \cdot n = 5 \cdot T(\frac{n}{2}) + \Theta(n)$

We guess that $T(n) \leqslant dn^{\lg 5} - d'n$ and use substitution-method.

- Induction Basis: $T(1) = c_1 \leqslant d \cdot 1^{\lg 5} - d' \cdot 1$, so long as $d - d' \geqslant c_1$
- Inductive Step: $T(n) = 5 \cdot T(\frac{n}{2}) + c_2 \cdot n \leqslant 5(d(\frac{n}{2})^{\lg 5} - d'(\frac{n}{2})) + c_2 n = dn^{\lg 5} - (\frac{5}{2}d' - c_2)n \leqslant dn^{\lg 5} - d'n$, so long as $\frac{3}{2}d' \geqslant c_2$

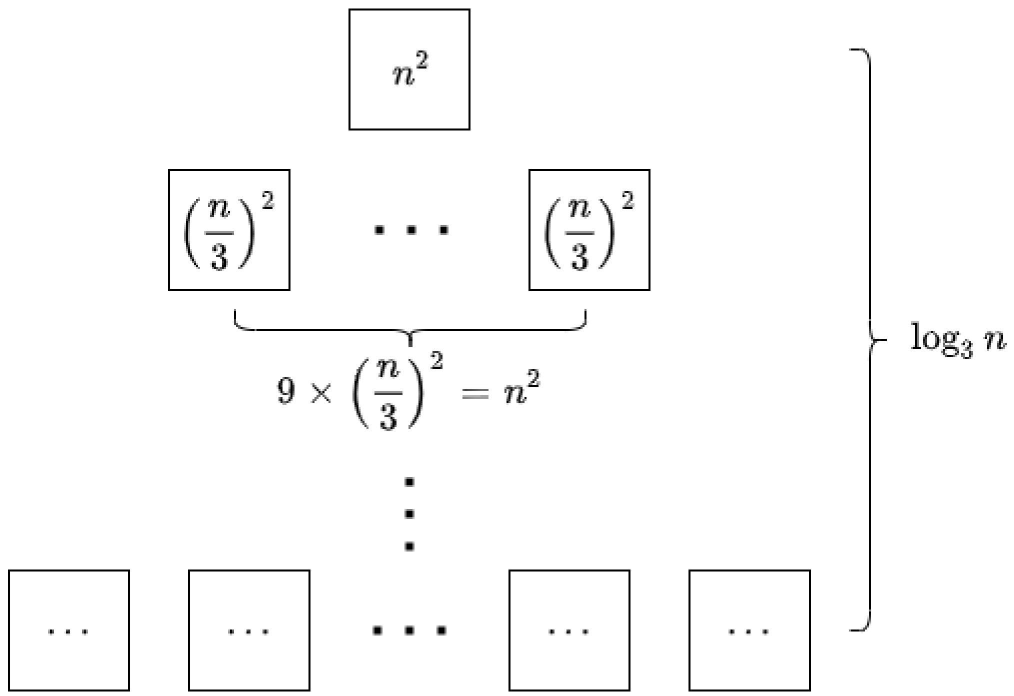
So $T(n) = O(n^{\lg 5})$

Algorithm B:

- Let $T(n)$ be the runtime of B on instance of size n
- Clearly, $T(1) = c = \theta(1)$
- $T(n) = 2 \cdot T(n-1) + c = \sum_{i=1}^{n} 2^{n-1} c = (2^n - 1)c = O(2^n)$

Algorithm C:

- Let $T(n)$ be the runtime of C on instance of size n
- $T(n) = 9 \cdot T(\frac{n}{3}) + c \cdot n^2$



So $T(n) = 9 \cdot T(\frac{n}{3}) + c \cdot n^2 = c \cdot n^2 \log_3 n = O(n^2 \log_3 n)$

Answer:

Obviously, the algorithm $T_B(n) = O(2^n)$ is much slower than $T_A(n) = O(n^{\lg 5})$ and $T_C(n) = O(n^2 \log_3 n)$, so we compare the latter two algorithms. Because $\lim\limits_{n \to \infty} \frac{n^2 \log_3 n}{n^{\log_2 5}} = 0$, we think the algorithm C is faster than algorithm A. So we choose algorithm C.

Problem 5

Overview:

We create an index array $I[1...n] = \{1, ..., n\}$, sort the origin array $A[1...n]$ with index array I, in the time complexity $O(n \log n)$. Then we remove the duplicates in A' and corresponding indices in I', in the time complexity $O(n \log n)$. Finally, we reorder the new array A'' by sorting the new index array I'' and get the final result.

Algorithm:

Let $A[1...n]$ be the origin array and $I[1...n] = \{1, ..., n\}$ be the index array.

Algorithm 6 Remove Duplicates

---

function MERGE(leftA, rightA, leftB, rightB)
  $m =$ length of leftA
  $m' =$ length of rightA

```
    solA[1...(m+m')], solB[1...(m+m')]
    i = 1, j = 1, k = 1
    while i <= m + m' do
        if k > m' or j <= m and leftA[j] <= rightA[k] then
            solA[i] = leftA[j]
            solB[i] = leftB[j]
            j = j + 1
        else
            solA[i] = rightA[k]
            solB[i] = rightB[k]
            k = k + 1
        end if
        i = i + 1
    end while
    return solA[1...(m+m')], solB[1...(m+m')]
end function
function MergeSort(A, B)
    if n == 1 then
        solA[1...n] = A[1...n]
        solB[1...n] = B[1...n]
    else
        leftSolA[1...(n/2)], leftSolB[1...(n/2)] = MergeSort(A[1...(n/2)], B[1...(n/2)])
        rightSolA[1...(n/2)], rightSolB[1...(n/2)] = MergeSort(A[(n/2+1)...n], B
        [(n/2+1)...n])
        solA[1...n] = Merge(leftSolA[1...(n/2)], rightSolA[1...(n/2)], leftSolB[1...(n/2)],
        rightSolB[1...(n/2)])
    end if
    return solA[1...n], solB[1...n]
end function
function RemoveDuplicates(A, I)
    if n == 1 then
        return A
    end if
    A', I' = MergeSort(A, I)
    A''[1...n], I''[1...n]
    length = n
    index = 1
    isDuplicated = false
    curr = A'[1]
    for i = 2 to n do
        if curr == A'[i] then
            if isDuplicated == false then
                length = length - 1
            end if
            length = length - 1
            isDuplicated = true
        else
            if isDuplicated == false then
                A''[index] = A'[i − 1]
                I''[index] = I'[i − 1]
```

```
            index = index + 1
        end if
        curr = A'[i]
        isDuplicated = false
    end if
end for
if isDuplicated == false then
    A''[index] = A'[n]
    I''[index] = I'[n]
end if
I''', A'''[1...length] = MergeSort(I''[1...length], A''[1...length])
return A'''[1...length]
end function
```

Time Complexity:

- **Merge:** $T_1(n) = c_1 + c_2 n = O(n)$
- **MergeSort:** $T_2(n) = 2T_2(\frac{n}{2}) + T_1(n) = T_1(n)\log n = O(n\log n)$
- **RemoveDuplicates:** $T_3(n) = T_2(n) + c_3 + c_4(n-1) + c_5 + T_2(n) = 2T_2(n) + c_4 n + (c_3 - c_4 + c_5) = O(n\log n)$

So the time complexity is $O(n\log n)$

Correctness:

We create an index array $I[1...n] = \{1, ..., n\}$, sort the origin array $A[1...n]$ with index array I, and then we get the sorted array A' and corresponding index array I'. Then we remove the duplicates in A' and corresponding indices in I', and get new array A'' with no duplicate and indices I''. Finally, we reorder the new array A'' by sorting the new index array I'' and get the final result array A''' with no duplicate and correct order.

Problem 6

(a) $(2,1), (3,1), (8,6), (8,1), (6,1)$

(b)

Answer:

The running time of insertion sort is $T(n) = c_1 n + c_2\left(\sum_{j=2}^{n} t_j\right) - c_2$ and $\sum_{j=2}^{n} t_j$ is the number of inversions in the input array.

Prove:

We know that swapping two adjacent elements in an array, provided that the former is greater than the latter, will only reduce the number of inversions by one. For example, $\{4,3,2,1\} \rightarrow \{4,2,3,1\}$, the number of inversions change from 6 to 5.

The only one swap operation (equivalently) in insertion sort is the 5-th line `A[i+1] = A[i]` and the 4-th line while-loop statement makes sure that `A[i]` is greater than `A[i+1]` before swapping. The 5-th line statement will be executed $\sum_{j=2}^{n} t_j$ times. It means that the number of inversions was reduced $\sum_{j=2}^{n} t_j$ times.

Finally, the array is sorted. We know that the number of inversions in the sorted array is zero, which means the number of inversions in the original array is $\sum_{j=2}^{n} t_j$. Proof completed.

(c)

Let A be the array.

Algorithm 7 CountInversions

```
function MERGE(left, right)
    m = length of left
    m' = length of right
    sol[1...(m+m')]
    i = 1, j = 1, k = 1
    count = 0
    while i <= m + m' do
        if k > m' or j <= m and left[j] <= right[k] then
            sol[i] = left[j]
            j = j + 1
        else
            sol[i] = right[k]
            count = count + k
            k = k + 1
        end if
        i = i + 1
    end while
    return sol[1...(m+m')], count
end function
function MERGESORT(A)
    if n == 1 then
        return A[1...n], 0
    else
        leftSol[1...(n/2)], leftCount = MergeSort(A[1...(n/2)])
        rightSol[1...(n/2)], rightCount = MergeSort(A[(n/2+1)...n])
```

```
        sol[1...n], mergeCount = Merge(leftSol[1...(n/2)], rightSol[1...(n/2)])
        return sol[1...n], leftCount + rightCount + mergeCount
    end if
end function
function CountInversions(A)
    A', count = MergeSort(A)
    return count
end function
```