Sample Solution for Problem Set 2

Data Structures and Algorithms, Fall 2021

September 23, 2021

Contents

1 Problem 1

In this problem, we assume that linked list L consists of an element $L.head$ representing the address of the head of linked list L.

(a)

A simple implementation is to update next pointer for each node.

Algorithm 1 Reversal

Require A linked list L;
Ensure A linked list with reversed order.
1: $prev = L.head, cur = prev \rightarrow next$
2: **while** $cur \neq NULL$ **do**
3: $\quad nxt = cur \rightarrow next$;
4: $\quad cur \rightarrow next = prev$;
5: $\quad prev = cur, cur = nxt$;
6: $L.head = prev$;

(b)

Suppose each nodes consist of both $next$ and $prev$ pointer. We will develop the following algorithms based on the fact $x.next = x.np \oplus x.prev$ for each node x. To simplify our algorithm, we may further assume that queries are all valid.

Algorithm 2 Exclusive-Or Linked List Insertion

Require A linked list L, a node x and an index i;
Ensure A linked list L with node x inserted at the i-th position.
1: $prev = 0, cur = L.head, cnt = 1$;
2: **while** $cnt < i$ **do**
3: $\quad nxt = cur \rightarrow np \oplus prev$;
4: $\quad prev = cur, cur = nxt$;
5: $\quad cnt = cnt + 1$;
6: $x.np = prev \oplus cur$
7: **if** $i == 1$ **then** $L.head = \&x$;
8: **if** $prev! = 0$ **then** $prev \rightarrow np = prev \rightarrow np \oplus cur \oplus \&x$;
9: **if** $cur! = 0$ **then** $cur \rightarrow np = cur \rightarrow np \oplus prev \oplus \&x$;

Algorithm 3 Exclusive-Or Linked List Deletion

Require A linked list L and an index i;
Ensure A linked list L with the i-th position removed.
1: $prev = 0, cur = L.head, cnt = 1$;
2: **while** $cnt < i$ **do**
3: $\quad nxt = cur \to np \oplus prev$;
4: $\quad prev = cur, cur = nxt$;
5: $\quad cnt = cnt + 1$;
6: $nxt = cur \to np \oplus prev$;
7: **if** $i == 1$ **then** $L.head = nxt$;
8: **if** $prev \neq 0$ **then** $prev \to np = prev \to np \oplus cur \oplus nxt$;
9: **if** $nxt \neq 0$ **then** $nxt \to np = nxt \to np \oplus prev \oplus cur$;

2 Problem 2

We may assume that queries are valid. Let S be the original stack(in order to distinguish operations on S from operations on max-stack, we will use $S.push$ and $S.pop$ in the following analysis).

Algorithm

- push(x): If S is nonempty, let $y = S.pop()$, then push y to the stack S, and lastly push $\{x, max(y.second, x)\}$ to the stack S; otherwise, push $\{x, x\}$ to the stack S.
- pop(): Let $y = S.pop()$, return $y.first$.
- max(): Let $y = S.pop()$, push y to the stack S, and lastly return $y.second$.

Correctness

All we have to verify is that $S.top().second = \max_{x \in S} x.first$. Let's prove it by induction.

- If we do the push operation, it can be seen that $\max(x, y.second)$ (or x if stack is empty before push operation) is exactly the maximum value among all remaining elements after that step by induction hypothesis.
- If we do the pop operation, correctness can be seen directly by induction hypothesis.

Time Complexity

Obviously, in the worst case, time complexity of above algorithm is $\Theta(1)$ per query, as we only do constant times of push or pop operation in each type of query.

Remark

- Algorithm will be scored according to correctness proof and time complexity mainly.
- and again, **Do NOT simulate stack via array!** Use $S.pop(), S.push(x), S.top(), S.empty, etc.$ instead.
- Please check corner cases like pushing duplicate elements in a row before you hand in your homework.
- You should not assume elements are integers. To be more specific, you should not assume that numeric operation like addition or subtraction exists.

3 Problem 3

We may assume that the input infix expression is valid. Let stk be a stack. Let string $infix$ be the input infix expression and use vector $postfix$ to store output postfix.

Overview

First, walk over all the characters of the input infix expression. For each character $infix[i]$, $0 \leq i < infix.length()$:

- If $infix[i]$ is a operand, push it back to the $postfix$.
- If $infix[i]$ is a operator, while S is not empty and the priority of stk's top element is higher than $infix[i]$, pop elements from the stack stk and push it back to the $postfix$ repeatedly. Then, push $infix[i]$ into stk.

Pop elements from the stack $stack$ and push it back to the $postfix$, until stk is empty. Then we got the desire output $postfix$.

Pseudocode

```
1  function Infix2postfix(infix):
2      vector postfix
3      Stack S
4      for i from 0 to infix.length() - 1 do
5          if '0' <= infix[i] and infix[i] <= '9' then
6              postfix.push_back(infix[i])
7          else
8              while !stk.empty() and stk.top() < infix[i] do
9                  postfix.push_back(stk.pop())
10             end while
11             stk.push(infix[i])
12         end if
13     end for
14     while !stk.empty() do
15         postfix.push_back(stk.pop())
16     end while
17     return postfix
```

Time Complexity

Each character in $infix$ will be go through at most once. Each operator will be push into stack and pop from stack at most once. The time complexity is $O(n)$.

4 Problem 4

a Since $T(n) = 5T(n/2) + O(n)$, we have $T(n) = O\left(\sum_{i=1}^{\log_2 n} \left(\frac{5}{2}\right)^{i-1}\right) = O(5^{\log_2 n}) = O(n^{\log_2 5})$

b Since $T(n) = 2T(n-1) + O(1)$, we have $T(n) = O(\sum_{i=1}^{n} 2^{i-1}) = O(2^n)$.

c Since $T(n) = 9T(n/3) + O(n^2)$, we have $T(n) = O(\sum_{i=1}^{\log_3 n} n^2) = O(n^2 \log_3 n)$

Analysis

Algorithm B runs in exponential time, so it is the slowest while n is suffiently large. Compare algorithm A and algorithm C. We have

$$\lim_{n\to\infty} \frac{n^{\log_2 5}}{n^2 \log_3 n} = \infty$$

Thus, $n^2 \log_3 n = O(n^{\log_2 5})$. For n can be arbitrarily large, choose algorithm C.

5 Problem 5

Because of the ambiguous description of this problem, I will give sample solution to the latest version, and give a brief introduction to harder version : solve the task and keep original order of the array.

Key idea for this task is to divide and conquer. We can divide the array $A[1...n]$ to two arrays $A[1...mid]$ and $A[mid+1...n]$, like Merge-Sort, and we solve two subproblems and get two sorted duplicate-removed subarrays. Now the only task is to combine these subarrays, this process is also similar to Merge-Sort, needs $O(n)$ time, n is the number of elements of the array on the subproblem.

5.1 Algorithm

Algorithm 4 Duplicate Removal (DR)

Global declaration $A[1...n], B[1...n]$
Require left boundary l, right boundary r.
Ensure k, the number of elements of duplicate-removed array B on $A[l...r]$, which stored in $A[l...l+k-1]$:

```
    if l == r then
2:      return 1
    mid ← (l + r) >> 1;
4:  leftk ← DR(a, l, mid)
    rightk ← DR(a, mid + 1, r)
6:  k1 = 1, k2 = 1, k = 0
    while k1 <= leftk and k2 <= rightk do
8:      if A[l + k1 − 1] < A[mid + k2] then
            B[l + k] ← A[l + k1 − 1]
10:         k ← k + 1
            k1 ← k1 + 1
12:     else
            B[l + k] ← A[mid + k2]
14:         k ← k + 1
            k2 ← k2 + 1
16: while k1 <= leftk do
        B[l + k] ← A[l + k1 − 1]
18:     k ← k + 1
        k1 ← k1 + 1
20: while k2 <= rightk do
        B[l + k] ← A[mid + k2]
22:     k ← k + 1
        k2 ← k2 + 1
24: k = 0
    for i ← 1 to leftk + rightk do
26:     if i = 1 or B[l + i − 1] ≠ B[l + i − 2] then
            A[l + k] ← B[l + i − 1]
28:         k ← k + 1
    return k
```

Time Complexity

As usual, we let the cutting point of subproblem on $A[L...R]$ is midpoint, $T(n) = 2T(n/2) + O(n) = O(nlogn)$.

Correctness

We use structured induction to prove the correctness.

For the subproblem on $A[L...R]$, we do nothing when $L = R$, it's clearly correct becuase it's duplicate-removed.

Otherwise we can solve two subproblems and get two duplicate-removed subarray $B1$ and $B2$, (note that elements of $B1$ or $B2$ may be not the permutation of $A1$ or $A2$, and the number of elements may be changed, but we can use the memory of $A[L...R]$ to store them.)The process of combination keeps the attribute, and return the array B (stored in $A[L...R]$) sorted and duplicate-removed. And the result of the entire array is that we want to get.

Brief introduction to other version

Sort the elements with using value as first keyword, scan the entire array and remove duplicates, and sort the rest elements with using origin position as first keyword. Time Complexity also meets the requirement. The extra memory we used depends on the sort you choose. It's necessary for you to learn about extra memory of different sorts.

6 Problem 6

Solution

(a) (1, 5), (2, 5), (3, 4), (3, 5), (4, 5)

(b) Assume that the number of inversions is m, then the running time of insertion sort is $\Theta(n+m)$.

Algorithm 5 INSERTION-SORT(A)

```
1: for j = 2 to A.length do
2:     key = A[j]
3:     i = j − 1
4:     while i > 0 and A[i] > key do
5:         A[i + 1] = A[i]
6:         i = i − 1
7:     A[i + 1] = key
```

Proof. Denote $f_i = |\{j : j < i \text{ and } A_j > A_i\}|$. Obviously $m = \sum_{i=1}^{n} f_i$.
line 1-3 costs $\Theta(n)$.
When $j = j_0$, according to the loop invariant, $A[1, \cdots, j_0 - 1]$ is sorted. All the numbers A_i in the original array which satisfies $i < j$ and $A_i > A_j$ now form a continuous segment in $A[j_0 - f(j_0), j_0 - 1]$. So the comparison in line 4 will be executed exactly $f(j_0) + 1$ times, and line 5-8 will be executed $f(j_0)$ times. In this part, the total running time is $\sum_{i=1}^{n}(c_2(f(j_0)+1) + c_3 f(j_0) + c_4 f(j_0)) = \Theta(n+m)$.
So the total complexity is $\Theta(n+m)$. □

(c)

Algorithm 6 COUNTING-INVERSION(A)

```
function MERGE(A, p, q, r)
    n_1 = q − p + 1
    n_2 = r − q
    Let L[1, ··· , n_1 + 1], R[1, ··· , n_2 + 1] be new arrays
    for i = 1 to n_1 do
        L[i] = A[p + i − 1]
    for j = 1 to n_2 do
        R[j] = A[q + j]
    L[n_1 + 1] = ∞
    R[n_2 + 1] = ∞
    i = j = 1
    inv_cnt = 0
    for k = p to r do
        if L[i] ≤ R[j] then
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            j = j + 1
            inv_cnt = inv_cnt + n_1 − i + 1
    return inv_cnt
function COUNTING-INVERSION(A, p, r)
    if p = r then
        return 0
    else
        q = ⌊(p+r)/2⌋
        l_cnt = COUNTING-INVERSION(A, p, q)
        r_cnt = COUNTING-INVERSION(A, q + 1, r)
        return l_cnt + r_cnt + MERGE(A, p, q, r)
```

Remark

- Please pay attention to the definition of inversion. It is represented by index instead of value. Many of you have make mistakes in this.
- It's inaccurate to say the running time of insertion sort is a constant times the number of inversions. Because the length of the array also provides a lower bound of the time complexity. You can say when the length is fixed, the running time is linear with the number of inversions. And it's better to describe running time(time complexity) with O-representation.
- The basic idea for counting inversions is the same as merge sort. First we divide the array into two halves, say $L[1, \cdots, n_1]$ and $R[1, \cdots, n_2]$. Then we call the subroutine and solve the L part and R part recursively to get the inversion number of L and R respectively and make them sorted. Now we only need to count inversions (i, j) where i comes from L and j from R. This can be calculated when merging two subarrays. Suppose we have merge the first $x - 1$ numbers in L and $y - 1$ numbers in R. If $L[x] > R[y]$, then $R[y]$ will form an inversion with all the numbers after $L[x]$ in L, which contributes $n_1 - (x - 1)$ to the answer. This way, all the inversions will be enumerated by the right part. The time complexity $T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n \lg n)$.

- If you have designed an algorithm you cannot see correctness from pseudocode directly, talk is cheap, show yourself the code.