Sample Solution for Problem Set 3

Data Structures and Algorithms, Fall 2019

October 30, 2020

Contents

1 Problem 1

This algorithm is selection sort.

SelectionSort(A)

```
1: for i = 1 to n − 1 do
2:     s ← i
3:     for j = i + 1 to n do
4:         if A[j] < A[s] then
5:             s ← j
6:     swap A[i] and A[s]
```

(a)

statement	cost	times
1	c_1	n
2	c_2	$n-1$
3	c_3	$\sum_{i=1}^{n-1} t_i$
4	c_4	$\sum_{i=1}^{n-1}(t_i-1)$
5	c_5	$0 \le t \le \sum_{i=1}^{n-1}(t_i-1)$
6	c_6	$n-1$

(b)

- $t_i = n - i$, t is depended on the elements of A.
- $\sum_{i=1}^{n-1} t_i = \sum_{i=1}^{n-1}(n-i) = \frac{n^2-n}{2}$
- $\sum_{i=1}^{n-1}(t_i - 1) = \sum_{i=1}^{n-1} t_i - (n-1) = \frac{n^2-3n+2}{2}$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3\left(\frac{n^2-n}{2}\right) + c_4\left(\frac{n^2-3n+2}{2}\right) + c_5 t + c_6(n-1) \\
&= \left(\frac{c_3}{2} + \frac{c_4}{2}\right) n^2 + \left(c_1 + c_2 - \frac{c_3}{2} - \frac{3c_4}{2} + c_6\right) n + (-c_2 + c_4 - c_6) + c_5 t
\end{aligned}
$$

- best-case: $t = 0$, worst-case: $t = \frac{n^2-3n+2}{2}$
- At best-case and worst-case, $T(n) = an^2 + bn + c = \Theta(n^2)$.

(c)

- Loop invariant: After the i-th loop, subarray $A[1...i]$ is sorted and $A[i]$ is the smallest element of A$[i...n]$.
- Proof:
  - Initialization: There is only one element, A[1], when $i = 1$.
  - Maintain: $A[i-1]$ is the smallest element of $A[i-1...n]$. After exchange, $A[i]$ is the smallest one of $A[i...n]$ and $A[1...i]$ is still sorted, when $i \leftarrow i + 1$.
  - Termination: $A[1...n]$ is sorted, when $i = n$.
- Correctness: Elements are exchanged only and $A[1...n]$ is sorted.

2 Problem 2

(a) $T(n) = c_1 + c_2(n+2) + c_3(n+1) = (c_2 + c_3)n + (c_1 + 2c_2 + c_3) = \Theta(n)$.

(b)

- Loop Invariant: After i-th loop, $y = \sum_{j=i}^{n} c_j x^{j-i}$.
- Proof:
  - Initialization:

$$\begin{aligned} y &= c_n + x * 0 \\ &= c_n = c_n x^0 \\ &= \sum_{j=i}^{n} c_j x^{j-i} \end{aligned}$$

    , when $i = n$.
  - Maintain:

$$\begin{aligned} y_{new} &= c_i + x * y_{old} \\ &= c_i + x * \sum_{j=i+1}^{n} c_j x^{j-(i+1)} \\ &= c_i + \sum_{j=i+1}^{n} c_j x^{j-i} \\ &= \sum_{j=i}^{n} c_j x^{j-i} \end{aligned}$$

    , when $i \leftarrow i - 1$.
  - Termination: $y = \sum_{j=0}^{n} c_j x^j$, when $i = 0$.
- Correctness: $y = \sum_{j=0}^{n} c_j x^j$ is equal to P(x).

3 Problem 3

(a) We know

$$\forall n > 0, 0 < \frac{f(n) + g(n)}{2} \leq \max(f(n), g(n)) < f(n) + g(n).$$

Therefore, by definition $\max(f(n), g(n)) \in \Theta(f(n) + g(n))$.

(b)

$$\lim \frac{(n+a)^b}{n^b} = 1.$$

Therefore, by definition $(n + a)^b \in \Theta(n^b)$.

(c) None.

4 Problem 4

$$
\begin{aligned}
1 = n^{1/\lg n} &\ll \\
\lg(\lg^* n) &\ll \\
\lg^* n = \lg^*(\lg n) &\ll \\
2^{\lg^* n} &\ll \\
\ln\ln n &\ll \\
\sqrt{\lg n} &\ll \\
\ln n &\ll \\
\lg^2 n &\ll \\
2^{\sqrt{2\lg n}} &\ll \\
(\sqrt{2})^{\lg n} &\ll \\
n = 2^{\lg n} &\ll \\
n\lg n = \lg(n!) &\ll \\
n^2 = 4^{\lg n} &\ll \\
n^3 &\ll \\
(\lg n)! &\ll \\
(\lg n)^{\lg n} = n^{\lg\lg n} &\ll \\
(3/2)^n &\ll \\
2^n &\ll \\
n\cdot 2^n &\ll \\
e^n &\ll \\
n! &\ll \\
(n+1)! &\ll \\
2^{2^n} &\ll \\
2^{2^{n+1}}
\end{aligned}
\tag{1}
$$

5 Problem 5

We may assume that queries are valid. Let S, T be two stacks.

Algorithm

- Enqueue(x): Push x to stack S.
- Dequeue(x): If T is empty, pop element from stack S and push it to stack T repeatedly, until stack S is empty. After above operations, T is non-empty, and we will pop element from T, and return it.

Correctness

We will show that following condition holds after each type of query.

- Element x arrives earlier than element y iff x is in stack T and y is in stack S, or x, y are both in stack S but y is on the top of x, or x, y are both in stack T but x is on the top of y.

Note that stack T is nonempty before we pop element from T in each dequeue operation, we can imediately see that top element in stack T arrives earliest among all remaining elements, which shows the correctness of our algorithm. Therefore, the only thing we need to prove is above property holds after query.

Base case is obvious and let's move on to the induction step. If we enqueue element x, above condition still holds as x is on the top of stack S, and all other elements remain the same. If T is empty before we do the dequeue operation, condition still holds after the first half of that operation as what we do here is actually reversing the stack and cut it to T. The second half of dequeue operation maintains above property with almost the same reason in the first case.

Therefore, we prove the correctness of our algorithm.

Time Complexity

- Worst case: $\Theta(n)$ per query, where n is the remaining elements in the "queue".
- Amortized time complexity: $\Theta(1)$ per query. Note that each element will be pushed and popped constant times.

Remark

- Algorithm will be scored according to correctness proof mainly. You don't have to achieve $\Theta(1)$ amortized complexity to get full points.
- **Do NOT simulate stack via array!** Use $S.pop(), S.push(x), S.top(), S.empty, etc.$ instead.
- Please be cautious about variants in condition statement of for-loop, especially when you have something to do with it in this loop implicitly.
- If you have implemented something you cannot see correctness from pseudocode directly, please at least leave some comments on your implementation.

6 Problem 6

We may assume that queries are valid. Let S be the original stack(in order to distinguish operations on S from operations on min-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 $\{x, min(y.second, x)\}$ to the stack S; otherwise, push $\{x, x\}$ to the stack S.
- pop(): Let $y = S.pop()$, return $y.first$.
- min(): Let $y = S.pop()$, push y to the stack S, and lastly return $y.second$.

Correctness

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

- If we do the push operation, it can be seen that $\min x, y.second$ (or x if stack is empty before push operation) is exactly the minimum 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 operation like addition exists.

7 Problem 7

Algorithm

The data structure contains an array $A[]$ and an integer $size$. $A[]$ is initialized to empty and $size$ is initialized to 0.

Algorithm 1 add(x)

$A[size] \leftarrow x$
$size \leftarrow size + 1$

Algorithm 2 remove()

$i \leftarrow random(x) - 1$
$size \leftarrow size - 1$
swap($A[i]$,$A[size]$)
return $A[size]$

Correctness

We claim that the data structure maintains the following invariance:

invariance:$\{A[0], A[1], ..., A[size - 1]\}$ are all the elements in the queue.

initialization: $size = 0$ and $queue$ is empty, which is true.

maintaining: After function add, we plus $size$ by 1. Since $\{A[0], ..., A[size - 2]\}$ are elements before we add x to queue, and we set $A[size - 1] = x$, $\{A[0], ..., A[size - 1]\}$ are elements in queue. After function $remove$, the $\{A[0], ..., A[size - 1]\}$ are exactly all the elements in queue except $A[i]$.

To prove that we always return a uniform random element in $remove$ procedure, notice that $A[size]$ is equal to $A[i]$ in the queue, and each element in the queue has the same probability to be $A[i]$ according to the property of $random$.

Complexity

It is trivially $O(1)$ for both add and $remove$.

8 Problem 8

We first give an algorithm to solve the general problem of converting an infix expression to postfix expression. Denote the string of the infix expression as $A[]$ and n is the size of the string.

Algorithm

Create stack S initialized to empty. Set $pri[!] > pri[x] > pri[+]$.

1: **for** $i = 0$ **to** n **do**
2: **if** $A[i]$ is digit **then** Print(A[i])
3: **else**
4: **while** S is not empty and $pri\,[S.top()] \geq pri\,[A[i]]$ **do** $Print\,(S.pop())$
$S.push(A[i])$
5: **while** S is not empty **do** $Print\,(S.pop())$

Correctness

We prove it by induction. Induction hypothesis: the algorithm will turn infix expression to sufix expression correctly when the length of input is at most n.

When $n = 1$, the input is a single digit, the output is correct.

When $n > 1$, denote the last operator with the smallest priority as $A[x]$. (When there are three operator $!x+$, that means $A[x]$ is the last $+$ in the expression). Then $A[x]$ should be in the last place in postfix expression. That is true in our algorithm, since when we add $A[x]$ to the stack, it will pop all elements in stack out (because it has the smallest priority), so it will fall into the bottom of the stack. And nothing will pop it out until the fifth line of our algorithm, which will add $A[x]$ to the last place in the postfix expression.

Now consider $A[0...x-1]$ and $A[x+1...n]$ (might be empty string). They both have length smaller than n. Before $A[x]$ is push into the stack, the procedure can be seen as running our algorithm in $A[0...x-1]$, since $A[x]$ will pop all elements out, which is just like what we do in line 5. Thus, according to induction hypothesis, $A[0...x-1]$ will be turned to postfix expression correctly. $A[x+1...n]$ can also be seen as running our algorithm independently, since all elements in $A[0....x-1]$ is not in the stack, and $A[x]$ is in the bottom of the stack while no one popping it out. Thus, the final string is $postfix(A[0...x-1]) + postfix(A[x+1...n]) + A[x]$, which is correct.

Complexity

Each operator will be popped and pushed at most twice, and each digit will be looped at most once. The complexity is $O(n)$.

Remark

For this special problem (with constant type of operators), there is another algorithm to solve the problem. Simply shift the first $+$ to the last place, and shift the first x between each two $+$ to the position before the $+$. That algorithm runs in $O(cn)$ while there are c types of operators. It is not efficient if c is $\omega(1)$. Many students use this algorithm which is fine for this problem, but the algorithm describe above is more universal.