CS 218, Spring 2017 Homework 2
Abhishek Kumar SRIVASTAVA April 19, 2017
Student ID: 861307778

I certify that this submission represents my own original work

---

Solution 1. Given that

$$T(n) = \begin{cases} 2 & n = 2 \\ 4 \cdot T(n^{1/2}) + \log^2(n) & n > 2 \end{cases}$$

Consider when $n > 2$

$$T(n) = 4 \cdot T(n^{1/2}) + \log^2(n)$$

Master theorem cannot be applied on this equation. We need to apply transformation to convert into Master theorem equation.

We can substitute $n = 2^m$ in the equation. Which will give us

$$\begin{aligned} T(2^m) &= 4 \cdot T(2^{m/2}) + \log^2(2^m) \\ &= 4 \cdot T(2^{m/2}) + m^2 \end{aligned}$$

Substituting again in this equation by $T(2^m) = S(m)$. We get,

$$S(m) = 4 \cdot S(m/2) + m^2$$

Now we can apply Master theorem on the above equation.

Case 1: $f(n) \; \epsilon \; O(n^{\log_b a - \varepsilon})$ where $\varepsilon > 0$.

$m^2 \; \epsilon \; O(m^{\log_2 4 - \varepsilon}) = O(m^{2-\varepsilon})$

For any $\varepsilon > 0$ this is not possible. So case 1 fails.

Case 2: $f(n) \; \epsilon \; \Theta(n^{\log_b a} \cdot \log^k n)$ where $k \geq 0$.

$m^2 \; \epsilon \; \Theta(m^{\log_2 4} \cdot \log^k m) = \Theta(m^2 \cdot \log^k m)$

$k = 0$ satisfy the equation. So case 2 holds. Thats why,

$$\begin{aligned} S(m) &= \Theta(m^{\log_2 4} \cdot \log^{k+1} m) \\ &= \Theta(m^2 \cdot \log m) \end{aligned}$$

By substituting back $m = \log n$ we get,

$$\boxed{T(n) = \Theta(\log^2 n \cdot \log \log n)}$$

Solution 2. Given $A[1, \ldots, n]$ as a fixed array of distinct integers in which we have to find the position for the elements in array $X[1, \ldots, k]$.

Naive Solution:

We can search the positions of each element in $X[1, \ldots, k]$ by doing linear search in array $A[1, \ldots, n]$ for that element. Which will the complexity of

$$T(n) = O(n \cdot k)$$

But this is not the lower bound.

Binary Decision Tree Solution:

We can do better than the naive solution to get the lower bound by using binary decision tree model. We first sort the array $A[1, \ldots, n]$ and then do a binary search for each element in the array $X[1, \ldots, k]$.

Total cost of doing search this way will be cost of sorting the array and searching each element using binary search. Therefore,

$$T(n) = n \cdot \log n + k \cdot logn$$

$(n \cdot \log n)$ since it is the lower bound of sorting $A[1, \ldots, n]$ using comparison model.
$(k \cdot \log n)$ since $X[1, \ldots, k]$ contains k elements and doing binary search on the sorted array $A[1, \ldots, n]$ have the lower bound of $\log n$.

$$\therefore T(n) = O((n + k) \log n)$$

This gives the lower bound on the time complexity, as a function of n and k, using the binary decision tree model for searching and sorting.

Pseudocode for described algorithm is as following:

Array $A[1, \ldots, n]$ ▷ Fixed Array
Array $X[1, \ldots, k]$ ▷ Array to be searched
Array $I[1, \ldots, k]$ ▷ Array to store position

function FIND_POSITION(A, X, I)

  SORT(A) ▷ Sort the array A in increasing order

  for i less than length(X) **do**
    $I[i]$ = BINARY_SEARCH($A, X[i]$) ▷ Search $X[i]$ in A and return its position
  end for

end function

Solution 3. Implement a queue using two stacks S_1 and S_2.

Part 1:

Queue is data structure which follows **FIFO**(First In First Out) property. On the other hand Stack is data structure which follows **LIFO**(Last In First Out) property. Therefore to implement Queue using Stack we need two of them. We always treat one stack for en-queuing data and other one for de-queuing.

While ENQUEUE(x) operation we push the element x in one stack for example $S1$. For DEQUEUE(x) operation we first check that other stack($S2$) is empty or not. If it is not empty we pop the top element from $S2$ stack. If it is empty we then transfer all the elements from $S1$ to $S2$ and then perform the pop operation. By using this method we can achieve **FIFO** property. Psuedocode for the above described method is following:

```
Stack S1                                                   ▷ Input stack
Stack S2                                                   ▷ Output stack

function ENQUEUE(x)
    S1 · push(x)                                           ▷ Pushing in the stack S1
end function

function DEQUEUE()
    if S2.empty() = true then                              ▷ Check if S2 is empty
        TRANSFER_ELEM_S1_TO_S2()                           ▷ Transfer all elements from S1 to S2
    end if
    return S2.pop()                                        ▷ Pop the element from S2
end function

function TRANSFER_ELEM_S1_TO_S2()
    while S1.empty() = false do                            ▷ While S1 is not empty
        S2.push(S1.pop())                                  ▷ Pop elements from S1 and push in S2
    end while
end function
```

Part 2: Accounting Method

Since we are using two stack we will have 4 operation to perform. push and pop for both $S1$ and $S2$.

- **Proposed charging Scheme** :
  - For ENQUEUE(x) operation we will charge \$4
    - \$1 pays for the pushing the element in $S1$.
    - \$3 is deposited(credit invariant) on the element to pay \$1 when popping the element from $S1$ later.
    - \$1 pays for pushing the element in $S2$ from remaining \$2.
    - \$1 pays for popping the element from $S2$ from the last remaining \$1.
  - For DEQUEUE() operation we will charge \$0

For any sequence of n For ENQUEUE(x) and n DEQUEUE() operations the cost will be

$$T(n) = (4 + \ldots \text{n times} \cdots + 4) + (0 + \ldots \text{n times} \cdots + 0) = 4n$$

$\therefore$ Each operation costs

$$\boxed{\mathrm{T}(n) = 4n/2n = 2 = \mathrm{O}(1)}$$