CSE 373: Homework 2 solutions

Problem 1. (Worth 6 points)
Base case: For $n = 1$, note that

$$\sum_{i=1}^{1}(i+1) = 2 = \frac{1 \cdot (1+3)}{2}$$

Hence, the base case holds.

Induction hypothesis: Assume for some k that

$$\sum_{i=1}^{k}(i+1) = \frac{k(k+3)}{2}$$

Inductive step: Now consider the sum to $k+1$,

$$\begin{aligned}
\sum_{i=1}^{k+1}(i+1) &= \sum_{i=1}^{k}(i+1) + ((k+1)+1) \\
&= \frac{k(k+3)}{2} + (k+2) \quad \text{by the induction hypothesis.} \\
&= \frac{k^2+5k+4}{2} \\
&= \frac{(k+1)(k+4)}{2}
\end{aligned}$$

as we wanted. So $\sum_{i=1}^{n}(i+1) = \frac{n(n+3)}{2}$ for all $n \geq 1$, by induction.

Problem 2. (Worth 7 points)

	100	$2n+5$	$\log_2 n$	$5n^2$	$n \log_2 n$
$3n+1$	Ω	Θ	Ω	O	O
$0.001 * 2^{n-10}$	Ω	Ω	Ω	Ω^*	Ω
$\log_{10} n^n$	Ω	Ω	Ω	O	Θ^*

To see that $0.001 \cdot 2^{n-10}$ is $\Omega(5n^2)$, simply notice that the first expression is an exponential, while the second expression is a polynomial. And we know that any exponential (with a base greater than 1) grows faster than any polynomial.

For the other box, notice that $\log_{10} n^n = n \log_{10} n = \frac{1}{\log_2 10} n \log_2 n$. Since the two expressions differ only by a constant factor, we have that $\log_{10} n^n$ is $\Theta(n \log_2 n)$.

Problem 3. (Worth 15 points, 8 points for the pseudocode)
There are several reasonable answers. I'll go through two, partly to give practice thinking about the runtime of algorithms.

Solution 1: Here, we'll use `Sa` to store the items of the queue (and we'll maintain that the most-recently added item is always put at the bottom of the stack). The stack `Sb` will be used as

temporary storage when we need to reverse the order of the items.

```
isEmpty():  Boolean {
    return Sa.isEmpty()
}

dequeue():  Object {
    if Sa is empty, then return an error message
    else return Sa.pop()
}

enqueue(item:  Object) {
    //Reverse the order of the items, moving them to Sb
    while Sa is not empty do {
        Sb.push( Sa.pop() )
    }
    Sb.push(item)
    //Now put the items back the way they were, moving them to Sa
    while Sb is not empty do {
        Sa.push( Sb.pop() )
}
```

(b) The method `isEmpty` simply checks whether the stack `Sa` is empty. So it takes $O(1)$ time. The method `dequeue` checks whether `Sa` is empty, then pops an item. So again, this is $O(1)$ time. The method `enqueue` must move all n items from one stack to another, then back again. So this takes $O(n)$ time.

(c) In the worst case, we first enqueue all n items, then dequeue them all. Each dequeue takes constant time, hence the entire dequeueing will take $O(n)$ time. Enqueueing, however, takes longer. The first enqueue will take time proportional to 1, the number of items in the queue. The second enqueue takes time proportional to 2, the third takes time proportional to 3, and so on up to n. Hence, the total time to enqueue is proportional to

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

So the total time to enqueue and dequeue in the worst case is $O(n^2)$. (You can see that this is actually the worst time possible since each enqueue can never take more than $O(n)$ time, and each dequeue is always $O(1)$ time. So at worst, we still can't take any longer than $O(n^2)$ time.)

By the way, notice that if we alternate enqueueing and dequeueing, then the total time will be just $O(n)$, since we'll always enqueue onto an empty queue, which takes constant amount of time per enqueue.

Solution 2: Here, we'll be in one of two states. In the first state, `Sa` will contain all the items in the queue with the most recently added item on the top of `Sa`. The stack `Sb` will be empty.

In the other state, `Sa` will be empty, while the stack `Sb` will contain all the items in queue with the most recently added item on the bottom of `Sb`.

Notice that in the first state, we can enqueue simply by pushing the item onto `Sa`. In the second state, we can dequeue simply by popping off the item from `Sb`. We can move between the two states by popping all the items off of one stack and pushing them onto the other.

```
isEmpty():  Boolean {
    return (Sa.isEmpty() and Sb.isEmpty())
}

dequeue():  Object {
    //If we're in the first state, then move into the other state
    if Sb is empty, then {
        while Sa is not empty do {
            Sb.push( Sa.pop() )
        }
    }
    if Sb is empty, then return an error message
    else return Sa.pop()
}

enqueue(item:  Object) {
    //If we're in the second state, then move into the first
    if Sa is empty, then {
        while Sb is not empty do {
            Sa.push( Sb.pop() )
        }
    }
    Sa.push(item)
}
```

(b) The method `isEmpty` simply checks whether the stack `Sa` is empty. So it takes $O(1)$ time. The method `dequeue` in the worst case moves all of the items from one stack to the other. So this takes $O(n)$ time. Similarly, the method `enqueue` may have to move all n items from one stack to another, taking $O(n)$ time.

(c) In the worst case, we first enqueue $n/2$ items. We then alternate enqueueing and dequeueing $n/2$ times. We then finish by dequeueing the $n/2$ items.

To see what the running time is, first notice that for the first $n/2$ enqueues, we are always in the first state. So we never have to move from one state to another. Hence, each of these enqueues takes $O(1)$ time. This means that the first $n/2$ enqueues take $O(n)$ time. Now, when we alternate between enqueueing and dequeueing, we need to move from the first state to the other. So each of these operations takes time proportional to the number of items in the queue— that is, $n/2$. Hence, the alternating enqueueing and dequeuing takes time proportional to $n^2/2$, which is $O(n^2)$. Finally, during the last $n/2$ dequeues, we'll always be in the second state. So each dequeue will take time $O(1)$, for a total of $O(n)$ time to dequeue the $n/2$ items. Hence, the total time of these operations is $O(n^2)$. (It is not hard to see that it can be no worse than this— each enqueue and dequeue takes time at most $O(n)$. So at worst, the total time can't be more than $O(n^2)$.)

By the way, notice that if we first enqueue n items, then dequeue them all, it takes just $O(n)$ time (unlike in solution 1). The reason for this is that when we enqueue, we'll remain in the first state, so we'll never need to move all of the items from one stack to another. Hence, each enqueue takes just $O(1)$ time. Then when we dequeue, the we'll have to move to the second state initially, taking $O(n)$ time. But then, we remain in the second state, so we'll never have to swap all of the items again. Hence, each dequeue past the first takes $O(1)$ time. This is a total of $O(n)$ time.