Problem Set 1

All parts are due on September 13, 2019 at 6PM. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on `alg.mit.edu`.

Problem 1-1. [20 points] **Asymptotic behavior of functions**

For each of the following sets of five functions, order them so that if f_a appears before f_b in your sequence, then $f_a = O(f_b)$. If $f_a = O(f_b)$ and $f_b = O(f_a)$ (meaning f_a and f_b could appear in either order), indicate this by enclosing f_a and f_b in a set with curly braces. For example, if the functions are:

$$f_1 = n, \qquad f_2 = \sqrt{n}, \qquad f_3 = n + \sqrt{n},$$

the correct answers are $(f_2, \{f_1, f_3\})$ or $(f_2, \{f_3, f_1\})$.

Note: Recall that a^{b^c} means $a^{(b^c)}$, not $(a^b)^c$, and that $\log$ means $\log_2$ unless a different base is specified explicitly. Stirling's approximation may help for part **d)**.

a)	**b)**	**c)**	**d)**
$f_1 = (\log n)^{2019}$	$f_1 = \log((\log n)^{6006})$	$f_1 = 3^{4n}$	$f_1 = 2^n$
$f_2 = n^2 \log(n^{2019})$	$f_2 = \log \log n$	$f_2 = 3^{2^n}$	$f_2 = n^3$
$f_3 = n^3$	$f_3 = n^{6006} \log n$	$f_3 = 2^{2^{n+2}}$	$f_3 = \binom{n}{n/2}$
$f_4 = 2.019^n$	$f_4 = \log(6006^{n^{6006}})$	$f_4 = 4^{n^3}$	$f_4 = n!$
$f_5 = n \log n$	$f_5 = (\log n)^{\log(n^{6006})}$	$f_5 = 10^n$	$f_5 = \binom{n}{3}$

Solution:

a. $(f_1, f_5, f_2, f_3, f_4)$. This order follows from knowing that $\log$ grows slower than n^a for all $0 < a$, and n^a grows slower than n^b for $0 < a < b$, as well as following logarithm and exponentiation rules.

b. $(\{f_1, f_2\}, f_4, f_3, f_5)$. This order follows from elementary exponentiation and logarithm rules, converting some of the exponent bases to 2 and remembering that big-O is much more sensitive to changes in exponents.

c. $(f_5, f_1, f_4, f_2, f_3)$. This order follows after converting all the exponent bases to 2 and again remembering that big-O is much more sensitive to changes in exponents.

d. $(\{f_2, f_5\}, f_3, f_1, f_4)$. This order follows from the definition of the binomial coefficient and Stirling's approximation. The trickiest one is $f_3 = \Theta(2^n/\sqrt{n})$ (by repeated use of Stirling), which grows slower than f_1.

Rubric:

- 5 points per set for a correct order
- -1 point per inversion
- -1 point per grouping mistake, e.g., $(\{f_1, f_2, f_3\})$ instead of (f_2, f_1, f_3) is -2 points because they differ by two splits.
- 0 points minimum

Problem 1-2. [25 points] **Better Sequences**

In Lecture 2, we used both linked lists and dynamic arrays to support the sequence interface. The linked list presented supports $O(1)$-time `insert_first(x)` and `delete_first()` operations, while the dynamic array supports amortized $O(1)$-time `insert_last(x)` and `delete_last()`. This seems like a compromise. In this problem, you will extend each of these data structures to support all four first/last sequence operations in $O(1)$ time. (For any algorithm you describe in this class, including data structure operations, you should **argue that it is correct**, and **argue its running time**.)

(a) [5 points] Describe how to modify a linked list to store and maintain $O(1)$ additional information to support the `insert_last(x)` operation in worst-case $O(1)$ time.

Solution: In addition to maintaining a pointer to the head of the linked list, also maintain a pointer to the tail, i.e., the last node of the linked list. When inserting a new last item, `insert_last(x)` can then find the current last item from the stored tail pointer, construct and link the new item after, and then relink the tail pointer to maintain the invariant that the tail pointer points to the tail, all in $O(1)$ time. `delete_last(x)` remains $O(n)$ time as the tail does not know which node precedes it.

Rubric:

- 3 points for description of modification
- 1 point for argument of correctness
- 1 point for argument of running time
- Partial credit may be awarded

(b) [10 points] Describe how to modify a linked list to support each of the four first/last sequence operations in $O(1)$ time. Are your operation running times worst-case or amortized?
Hint: Store and maintain $O(1)$ additional information at each node.

Solution: In addition to storing a head and tail pointer as in part (a), store with each linked list node a next **and previous** pointer to the respective nodes after and before it in the sequence. Now, the last node stores a pointer to the node before it, so during a `delete_last()` operation, the data structure can identify the node before it in $O(1)$ time, and relink the tail pointer to maintain the invariant. Analogously to how next pointers are maintained during dynamic operations in the singly linked list presented in lecture, the previous pointers can be similarly maintained.

Rubric:

- 6 points for description of modification
- 2 point for argument of correctness
- 2 point for argument of running time
- Partial credit may be awarded

(c) [10 points] Describe how to modify a dynamic array to support each of the four first/last sequence operations in $O(1)$ time. Are your operation running times worst-case or amortized?

Hint: A dynamic array has fast last operations because it maintains additional space at the end of the array. Can you generalize?

Solution: Similar to the dynamic array presented in lecture, rebuild the sequence into a larger array whenever necessary; but instead of storing the items starting at index 0, store the sequence in the middle of the array, leaving linear extra space on both sides. For example, when we rebuild n elements, we can initialize a static array of size $2n$, with $n/2$ empty slots on either end. Then, we can keep track of the indices of the head and the tail item of the sequence stored in the array. Although rebuilding this array is a $\Theta(n)$ operation, rebuilding will not have to occur again until at least $O(n)$ elements have been inserted, each in $O(1)$ time. Thus, insertion at both first and last positions can be done in amortized $O(1)$ time. Similarly, rebuilding whenever the dynamic array contains two few items (e.g., $n/2$ items), ensures deletions at either end is amortized $O(1)$ time as well.

Rubric:

- 6 points for description of modification
- 2 point for argument of correctness
- 2 point for argument of running time
- Partial credit may be awarded

Problem 1-3. [10 points] Given a data structure that supports the four first/last sequence operations (`insert_first(x)`, `delete_first()`, `insert_last(x)`, `delete_last()`) each in $O(1)$ time, describe algorithms to implement the following higher-level operations in terms of the lower-level operations. Recall that `delete` operations return the deleted item. (Remember to argue **correctness** and **running time**!)

(a) [5 points] `shift_left(k)`: Move the first k items in order to the end of the sequence in $O(k)$ time. (At the end of the operation, the k^{th} item should be last and the $(k+1)^{\text{th}}$ item should be first.)

Solution:

To implement `shift_left(1)`, delete the first item and insert it into the last position in $O(1)$ time. The list maintains the relative ordering of all items in the sequence, except has moved the first item behind all the others, so `shift_left(1)` is correct. Then `shift_left(k)` is just the repeated application of `shift_left(1)`, so `shift_left(k)` is also correct and runs in $O(k)$ time.

```python
def shift_left(self, k):
    if (k < 1) or (k > len(self) - 1): return
    x = self.delete_first()
    self.insert_last(x)
    if k > 1: self.shift_left(k - 1)
```

Rubric:

- 3 points for description of algorithm
- 1 point for argument of correctness
- 1 point for argument of running time
- Partial credit may be awarded

(b) [5 points] `swap_ends()`: Swap the first item in the list with the last item in the list in $O(1)$ time.

Solution:

Swapping the first and last items in the list can be performed by simply deleting both ends in $O(1)$ time, and then inserting them back in the opposite order, also in $O(1)$ time. This algorithm is correct by the definitions of these operations.

```python
def swap_ends(self):
    x_first = self.delete_first()
    x_last  = self.delete_last()
    self.insert_first(x_last)
    self.insert_last(x_first)
```

Rubric:

- 3 points for description of algorithm
- 1 point for argument of correctness
- 1 point for argument of running time
- Partial credit may be awarded

Problem 1-4. [45 points] **Jen & Berry's**

Jen drives her ice cream truck to Linkoln Elementary at recess. All the kids rush to line up in front of her truck. Jen is overwhelmed with the number of students (there are $2n$ of them), so she calls up her associate, Berry, to bring his ice cream truck to help her out. Berry soon arrives and parks at the other end of the line of students. He offers to sell to the last student in line, but the other students revolt in protest: "The last student was last! This is unfair!"

The students decide that the fairest way to remedy the situation would be to have the back half of the line (the n kids furthest from Jen) reverse their order and queue up at Berry's truck, so that the last kid in the original line becomes the last kid in Berry's line, with the $(n+1)^{\text{th}}$ kid in the original line becoming Berry's first customer.

(a) [20 points] Given a linked list containing the names of the $2n$ kids, in order of the original line formed in front of Jen's truck (where the first node contains the name of the first kid in line), describe a $O(n)$-time algorithm to modify the linked list to reverse the order of the last half of the list. Your algorithm should not make any new linked list nodes or instantiate any new non-constant-sized data structures during its operation. (Remember to argue **correctness** and **running time**!)

Solution: Reverse the order of the last half of the nodes in a list in three stages:

- find the $(n-1)^{\text{th}}$ node a in the sequence (the end of Jen's line)
- for each node x from the n^{th} node b to the $(2n-1)^{\text{th}}$ node c, change the next pointer of x to point to the node before it in the original sequence
- change the next pointer of a and b to point to c and nothing respectively

Finding the $(n-1)^{\text{th}}$ node requires traversal next pointers $n-1$ times from the head of the list, which can be done in $O(n)$ time via a simple loop. We can compute n by halving the size of the list (which is guarenteed to be even).

To change the next pointers of the last half of the sequence, we can maintain pointers to the current node x and the node before it x_p, initially b and a respectively. Then, record the node x_n after x, relink x to point to the x_p, the node before x in $O(1)$ time. Then we can change the current node to x_n and the node before it to x, maintaining the desired properties for the next node to relink. Repeating n times, relinks all n nodes in the last half of the sequence in $O(n)$ time.

Lastly, by remembering nodes a, b, and c while the algorithm traverses the list, means that changing the exceptional next pointers at the front and back of the last half of the list takes $O(1)$, leading to an $O(n)$ time algorithm overall.

Rubric:

- 12 points for description of algorithm
- 4 point for argument of correctness
- 4 point for argument of running time
- Partial credit may be awarded

(b) [25 points] Write a Python function `reorder_students` that implements your algorithm. You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

Solution:

```python
def reorder_students(L):
    n = L.size // 2             # find the (n-1)th node
    a = L.head
    for _ in range(n - 1):
        a = a.next
    b = a.next                  # relink next pointers of last half
    x, x_p = b, a
    for _ in range(n):
        x_n = x.next
        x.next = x_p
        x_p, x = x, x_n
    c = x_p
    a.next = c                  # relink front and back of last half
    b.next = None
    return
```

Rubric:

- This part is automatically graded at `alg.mit.edu`.