Problem Set 1

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.

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:

\[
f_1 = \sqrt{n}, \quad f_2 = n, \quad f_3 = n + \sqrt{n},
\]

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 comparing factorials.

a) \( f_1 = \log(n^n) \)  \( f_2 = (\log n)^n \)  \( f_3 = \log(n^{6006}) \)  \( f_4 = (\log n)^{6006} \)  \( f_5 = \log \log(6006n) \)

b) \( f_1 = 2^n \)  \( f_2 = 6006^n \)  \( f_3 = 4 \times 6006^n \)  \( f_4 = (\log 6006)^n \)  \( f_5 = 6006^{n^2} \)

c) \( f_1 = n \)  \( f_2 = \sqrt{n} \)  \( f_3 = n \)  \( f_4 = n/6 \)  \( f_5 = n^6 \)

d) \( f_1 = n^4 + n! \)  \( f_2 = 2^{\log(n\sqrt{n})} \)  \( f_3 = 2^{(n-6)} \)  \( f_4 = 4^{3n \log n} \)  \( f_5 = 7^n \)

Solution:

a. \((f_5, f_3, f_4, f_1, f_2)\). Using exponentiation and logarithm rules, we can simplify these to \( f_1 = \Theta(n \log n) \), \( f_2 = \Theta((\log n)^n) \), \( f_3 = \Theta(\log n) \), \( f_4 = \Theta((\log n)^{6006}) \), and \( f_5 = \Theta(\log \log n) \). For \( f_2 \), note that \((\log n)^n > 2^n \) for large \( n \), so this function grows at least exponentially and is therefore bigger than the rest.

b. \((f_1, f_2, f_5, f_4, f_3)\). This order follows after converting all the exponent bases to 2. (For example, \( f_5 = 2^{\log((6006)^n)} \).) Remember that asymptotic growth is much more sensitive to changes in exponents: even if the exponents are both \( \Theta(f(n)) \) for some function \( f(n) \), the functions will not be the same asymptotically unless their exponents only differ by a constant.
c. \((\{f_2, f_5\}, f_4, f_1, f_3\})\). This order follows from the definition of the binomial coefficient and Stirling’s approximation. \(f_2\) has most terms cancel in the numerator and denominator, leaving a polynomial with leading term \(n^6/6!\). The trickiest one is \(f_4 = \Theta((6/(5^{5/6}))^n/\sqrt{n})\) (by repeated use of Stirling), which is about \(\Theta(1.57^n/\sqrt{n})\). Thus \(f_4\) is bigger than the polynomials but smaller than the factorial and \(n^n\). \(f_3 = \Theta(\sqrt{n}(6n/e)^{6n})\) which grows asymptotically faster than \(n^n\) by a factor of \(\Theta(n^{5n+(1/2)}(6/e)^{6n})\). (Originally, \(f_3\) was presented as \(6n!\) which could reasonably be interpreted as \(6(n!)\), which would put \(f_3\) before \(f_1\) in the order. Because of this, graders should accept \(f_1\) and \(f_3\) in either order.)

d. \((f_5, f_2, f_1, f_3, f_4)\). It is easiest to see this by taking the logarithms of the functions, which give us \(\Theta(n \log n), \Theta(\sqrt{n} \log n), \Theta(n \log n), \Theta(n^2), \Theta(\log n)\) respectively. However, asymptotically similar logarithms do not imply that the functions are asymptotically the same, so we consider \(f_1\) and \(f_3\) further. Note that \(f_4 = (4^\log n)^{3n} = (n^{\log 4})^{3n} = n^{6n}\). This is bigger (by about a factor of \(n^{5n}\)) than the larger of \(f_1\)’s terms, so \(f_3\) is asymptotically larger.

**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. [16 points] Given a data structure $D$ that supports Sequence operations:

- $D$.build($X$) in $O(n)$ time, and
- $D$.insert_at($i$, $x$) and $D$.delete_at($i$), each in $O(\log n)$ time,

where $n$ is the number of items stored in $D$ at the time of the operation, describe algorithms to implement the following higher-level operations in terms of the provided lower-level operations. Each operation below should run in $O(k \log n)$ time. Recall, $\text{delete}_at$ returns the deleted item.

**(a) reverse($D$, $i$, $k$):** Reverse in $D$ the order of the $k$ items starting at index $i$ (up to index $i + k - 1$).

**Solution:** Thinking recursively, to reverse the $k$-item subsequence from index $i$ to index $i + k - 1$, we can swap the items at index $i$ and index $i + k - 1$, and then recursively reverse the rest of the subsequence. As a base case, no work needs to be done to reverse a subsequence containing fewer than 2 items. This procedure would then be correct by induction.

It remains to show how to actually swap items at index $i$ and index $i + k - 1$. Note that removing an item will shift the index values at all later items. So to keep index values consistent, we will $\text{delete}_at$ the later index $i + k - 1$ first (storing item as $x_2$), and then $\text{delete}_at$ index $i$ (storing item as $x_1$). Then we insert them back in the opposite order, $\text{insert}_at$ item $x_2$ at index $i$, and then $\text{insert}_at$ item $x_1$ at index $i + k - 1$. This swap is correct by the definitions of these operations.

The swapping sub procedure performs four $O(\log n)$-time operations, so occurs in $O(\log n)$ time. Then the recursive reverse procedure makes no more than $k/2 = O(k)$ recursive calls before reaching a base case, doing one swap per call, so the algorithm runs in $O(k \log n)$ time.

```python
def reverse(D, i, k):
    if k < 2:  # base case
        return
    x2 = D.delete_at(i + k - 1)  # swap items i and i + k - 1
    x1 = D.delete_at(i)
    D.insert_at(i, x2)
    D.insert_at(i + k - 1, x1)
    reverse(D, i + 1, k - 2)  # recurse on remainder
```

**Rubric:**

- 5 points for description of algorithm
- 1 point for argument of correctness
- 2 point for argument of running time
- Partial credit may be awarded
(b) \(\text{move}(D, i, k, j)\): Move the \(k\) items in \(D\) starting at index \(i\), in order, to be in front of the item at index \(j\). Assume that expression \(i \leq j < i + k\) is false.

**Solution:** Thinking recursively, to move the \(k\)-item subsequence starting at \(i\) in front of the item at index \(j\), it suffices to move the item at index \(i\) in front of the item \(B\) at index \(j\), and then recursively move the remainder (the \((k - 1)\)-item subsequence starting at index \(i\) in front of \(B\)). As a base case, no work needs to be done to move a subsequence containing fewer than 1 item. If we maintain that: \(i\) is the index of the first item to be moved, \(k\) is number of items to be moved, and \(j\) denotes the index of the item in front of which we must place items, then this procedure will be correct by induction.

Note that after removing the item \(A\) at index \(i\), if \(j > i\), item \(B\) will shift down to be at index \(j - 1\). Similarly, after inserting \(A\) in front of \(B\), item \(B\) will be at an index that is one larger than before, while the next item in the subsequence to be moved will also be at a larger index if \(i > j\). Maintaining these indices then results in a correct algorithm.

This recursive procedure makes no more than \(k = O(k)\) recursive calls before reaching a base case, doing \(O(\log n)\) work per call, so the algorithm runs in \(O(k \log n)\) time.

```python
def move(D, i, k, j):
    if k < 1:
        return
    x = D.delete_at(i)
    if j > i:
        j = j - 1
    D.insert_at(j, x)
    j = j + 1
    if i > j:
        i = i + 1
    move(D, i, k - 1, j)
```

**Rubric:**
- 5 points for description of algorithm
- 1 point for argument of correctness
- 2 point for argument of running time
- Partial credit may be awarded
Problem 1-3. [20 points]  Binder Bookmarks

Sisa Limpson is a very organized second grade student who keeps all of her course notes on individual pages stored in a three-ring binder. If she has \( n \) pages of notes in her binder, the first page is at index 0 and the last page is at index \( n - 1 \). While studying, Sisa often reorders pages of her notes. To help her reorganize, she has two bookmarks, \( A \) and \( B \), which help her keep track of locations in the binder.

Describe a database to keep track of pages in Sisa’s binder, supporting the following operations, where \( n \) is the number of pages in the binder at the time of the operation. Assume that both bookmarks will be placed in the binder before any shift or move operation can occur, and that bookmark \( A \) will always be at a lower index than \( B \). For each operation, state whether your running time is worst-case or amortized.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>build(( X ))</td>
<td>Initialize database with pages from iterator ( X ) in ( O(</td>
</tr>
<tr>
<td>place_mark(( i, m ))</td>
<td>Place bookmark ( m \in {A, B} ) between the page at index ( i ) and</td>
</tr>
<tr>
<td></td>
<td>the page at index ( i + 1 ) in ( O(n) ) time.</td>
</tr>
<tr>
<td>read_page(( i ))</td>
<td>Return the page at index ( i ) in ( O(1) ) time.</td>
</tr>
<tr>
<td>shift_mark(( m, d ))</td>
<td>Take the bookmark ( m \in {A, B} ), currently in front of the page at</td>
</tr>
<tr>
<td></td>
<td>index ( i ), and move it in front of the page at index ( i + d ) for ( d \in {-1, 1} ) in ( O(1) ) time.</td>
</tr>
<tr>
<td>move_page(( m ))</td>
<td>Take the page currently in front of bookmark ( m \in {A, B} ),</td>
</tr>
<tr>
<td></td>
<td>and move it in front of the other bookmark in ( O(1) ) time.</td>
</tr>
</tbody>
</table>

**Solution:** There are many possible solutions. First note that the problem specifications ask for a constant-time \( \text{read}_\text{page}(i) \) operation, which can only be supported using array-based implementations, so linked-list approaches will be incorrect. Also note that that until both bookmarks are placed, we can simply store all pages in a static array of size \( n \), since no operations can change the sequence of pages until both bookmarks are placed. We present a solution generalizing the dynamic array we discussed in class. Another common approach could be to reduce to using two dynamic arrays (one on either end of bookmarks \( A \) and \( B \)), together with an array-based deque as described in Problem Session 1 to store the pages between bookmarks \( A \) and \( B \).

For our approach, after both bookmarks have been placed, we will store the \( n \) pages in a static array \( S \) of size \( 3n \), which we can completely re-build in \( O(n) \) time whenever \( \text{build}(X) \) or \( \text{place}_\text{mark}(i, m) \) are called (assuming \( n = |X| \)). To build \( S \):

- place the subsequence \( P_1 \) of pages from index 0 up to bookmark \( A \) at the beginning of \( S \),
- followed by \( n \) empty array locations,
- followed by the subsequence of pages \( P_2 \) between bookmarks \( A \) and \( B \),
- followed by \( n \) empty array locations,
- followed by the subsequence of pages \( P_3 \) from bookmark \( B \) to index \( n - 1 \).

We will maintain the separation invariant that \( P_1 \), \( P_2 \), and \( P_3 \) are stored contiguously in \( S \) with a non-zero number of empty array slots between them. We also maintain four indices with semantic
invariants: $a_1$ pointing to the end of $P_1$, $a_2$ pointing to the start of $P_2$, $b_1$ pointing to the end of $P_2$, and $b_2$ pointing to the start of $P_3$.

To support \texttt{read\_page}(i), there are three cases: either $i$ is the index of a page in $P_1$, $P_2$, or $P_3$.

- If $i < n_1$, where $n_1 = |P_1| = a_i + 1$, then the page is in $P_1$, and we return page $S[i]$.
- Otherwise, if $n_1 \leq i < n_1 + n_2$, where $n_2 = |P_2| = (b_1 - a_2 + 1)$, then the page is in $P_2$, and we return page $S[a_2 + (i - n_1)]$.
- Otherwise, $i > a_1 + n_2$, so the page is in $P_3$, and we return page $S[b_2 + (i - n_1 - n_2)]$.

This algorithm returns the correct page as long as the invariants on the stored indices are maintained, and returns in worst-case $O(1)$ time based on some arithmetic operations and a single array index lookup.

To support \texttt{shift\_mark}(m, d), move the relevant page at one of indices $(a_1, a_2, b_1, b_2)$ to the index location $(a_2 - 1, a_1 + 1, b_2 - 1, b_1 + 1)$ respectively, and then increment the stored indices to maintain the invariants. This algorithm maintains the invariants of the data structure so is correct, and runs in $O(1)$ time based on one array index lookup, and one index write. Note that this operation does not change the amount of extra space between sections $P_1$, $P_2$, and $P_3$, so the running time of this operation is worst-case.

To support \texttt{move\_page}(m), move the relevant page at one of indices $(a_1, b_1)$ to the index location $(b_1 + 1, a_1 + 1)$ respectively, and then increment the stored indices to maintain the invariants. If performing this move breaks the separation invariant (i.e., either pair $(a_1, a_2)$ or $(b_1, b_2)$ become adjacent), rebuild the entire data structure as described above. This algorithm maintains the invariants of the data structure, so is correct. Note that this algorithm: rebuilds any time the extra space between two adjacent sections closes; after rebuilding, there is $n$ extra space between adjacent sections; and the extra space between adjacent sections changes by at most one per \texttt{move\_page} operation. Thus, since this operation takes $O(n)$ time at most once every $n$ operations, and $O(1)$ time otherwise, this operation runs in \textbf{amortized} $O(1)$ time.

\textbf{Rubric:}

- 4 points for general description of a correct database
- 1 point for description of a correct \texttt{build}(X)
- 2 points for description of a correct \texttt{place\_mark}(i, m)
- 3 points for description of a correct \texttt{read\_page}(i)
- 2 points for description of a correct \texttt{shift\_mark}(m, d)
- 3 points for description of a correct \texttt{move\_page}(m)
- 1 point for analysis of running time for each operation (5 points total)
- Partial credit may be awarded
Problem 1-4. [44 points] Doubly Linked List

In Lecture 2, we described a singly linked list. In this problem, you will implement a doubly linked list, supporting some additional constant-time operations. Each node \( x \) of a doubly linked list maintains an \( x.\text{prev} \) pointer to the node preceding it in the sequence, in addition to an \( x.\text{next} \) pointer to the node following it in the sequence. A doubly linked list \( L \) maintains a pointer to \( L.\text{tail} \), the last node in the sequence, in addition to \( L.\text{head} \), the first node in the sequence. For this problem, doubly linked lists should not maintain their length.

(a) [8 points] Given a doubly linked list as described above, describe algorithms to implement the following sequence operations, each in \( O(1) \) time.

\[
\text{insert}\_\text{first}(x) \quad \text{insert}\_\text{last}(x) \quad \text{delete}\_\text{first}() \quad \text{delete}\_\text{last}()
\]

Solution: Below are descriptions of algorithms supporting the requested operations. Each of these algorithm performs each constant-sized task directly, so no additional argument of correctness is necessary. Each runs in \( O(1) \) time by relinking a constant number of pointers (and possibly constructing a single node).

\text{insert}\_\text{first}(x): Construct a new doubly linked list node \( a \) storing \( x \). If the doubly linked list is empty, (i.e., the head and tail are unlinked), then link both the head and tail of the list to \( a \). Otherwise, the linked list has a head node \( b \), so make \( a \)'s next pointer point to \( b \), make \( b \)'s previous pointer point to \( a \), and set the list’s head to point to \( a \).

\text{insert}\_\text{last}(x): Construct a new doubly linked list node \( a \) storing \( x \). If the doubly linked list is empty, (i.e., the head and tail are unlinked), then link both the head and tail of the list to \( a \). Otherwise, the linked list has a tail node \( b \), so make \( a \)'s previous pointer point to \( b \), make \( b \)'s next pointer point to \( a \), and set the list’s tail to point to \( a \).

\text{delete}\_\text{first}(): This method only makes sense for a list containing at least one node, so assume the list has a head node. Extract and store the item \( x \) from the head node of the list. Then set the head to point to the node \( a \) pointed to by the head node’s next pointer. If \( a \) is not a node, then we removed the last item from the list, so set the tail to None (head is already set to None). Otherwise, set the new head’s previous pointer to None. Then return item \( x \).

\text{delete}\_\text{last}(): This method only makes sense for a list containing at least one node, so assume the list has a tail node. Extract and store the item \( x \) from the tail node of the list. Then set the tail to point to the node \( a \) pointed to by the tail node’s previous pointer. If \( a \) is not a node, then we removed the last item from the list, so set the head to None (tail is already set to None). Otherwise, set the new tail’s next pointer to None. Then return item \( x \).

Rubric:

- 2 points for description and analysis of each correct operation
- Partial credit may be awarded
(b) [5 points] Given two nodes $x_1$ and $x_2$ from a doubly linked list $L$, where $x_1$ occurs before $x_2$, describe a constant-time algorithm to remove all nodes from $x_1$ to $x_2$ inclusive from $L$, and return them as a new doubly linked list.

**Solution:** Construct a new empty list $L_2$ in $O(1)$ time, and set its head and tail to be $x_1$ and $x_2$ respectively. To extract this sub-list, care must be taken when $x_1$ or $x_2$ are the head or tail of $L$ respectively. If $x_1$ is the head of $L$, set the new head of $L$ to be the node $a$ pointed to by $x_2$’s next pointer; otherwise, set the next pointer of the node pointed to by $x_1$’s previous pointer to $a$. Similarly, if $x_2$ is the tail of $L$, set the new tail of $L$ to be the node $b$ pointed to by $x_1$’s previous pointer; otherwise, set the previous pointer of the node pointed to by $x_2$’s next pointer to $b$.

This algorithm removes the nodes from $x_1$ to $x_2$ inclusive directly, so it is correct, and runs in $O(1)$ time by relinking a constant number of pointers.

**Rubric:**
- 3 points for description of a correct algorithm
- 1 point for argument of correctness
- 1 point for argument of running time
- Partial credit may be awarded

(c) [6 points] Given node $x$ from a doubly linked list $L_1$ and second doubly linked list $L_2$, describe a constant-time algorithm to splice list $L_2$ into list $L_1$ after node $x$. After the splice operation, $L_1$ should contain all items previously in either list, and $L_2$ should be empty.

**Solution:** First, let $x_1$ and $x_2$ be the head and tail nodes of $L_2$ respectively, and let $x_n$ be the node pointed to by the next pointer of $x$ (which may be None). We can remove all nodes from $L_2$ by setting both it’s head and tail to None. Then to splice in the nodes, first set the previous pointer of $x_1$ to be $x$, and set the next pointer of $x$ to be $x_1$. Similarly, set the next pointer of $x_2$ to be $x_n$. If $x_n$ is None, then set $x_2$ to be the new tail of $L$; otherwise, set the previous pointer of $x_n$ to point back to $x_2$.

This algorithm inserts the nodes from $L_2$ directly into $L$, so it is correct, and runs in $O(1)$ time by relinking a constant number of pointers.

**Rubric:**
- 4 points for description of a correct algorithm
- 1 point for argument of correctness
- 1 point for argument of running time
- Partial credit may be awarded

(d) [25 points] Implement the operations above in the `Doubly_Linked_List_Seq` class in the provided code template; do not modify the `Doubly_Linked_List_Node` class. You can download the code template including some test cases from the website.
Solution:

class Doubly_Linked_List_Seq:
    # other template methods omitted

def insert_first(self, x):
    new_node = Doubly_Linked_List_Node(x)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
    else:
        new_node.next = self.head
        self.head.prev = new_node
        self.head = new_node

def insert_last(self, x):
    new_node = Doubly_Linked_List_Node(x)
    if self.tail is None:
        self.head = new_node
        self.tail = new_node
    else:
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node

def delete_first(self):
    assert self.head
    x = self.head.item
    self.head = self.head.next
    if self.head is None:
        self.tail = None
    else:
        self.head.prev = None
    return x

def delete_last(self):
    assert self.tail
    x = self.tail.item
    self.tail = self.tail.prev
    if self.tail is None:
        self.head = None
    else:
        self.tail.next = None
    return x

def remove(self, x1, x2):
    L2 = Doubly_Linked_List_Seq()
    L2.head = x1
    L2.tail = x2
    if x1 == self.head:
        self.head = x2.next
    else:
        x1.prev.next = x2
    if x2 == self.tail:
        self.tail = x1.prev
    else:
        x2.next.prev = x1
    x1.prev = None
    x2.next = None
    return L2

def splice(self, x, L2):
    xn = x.next
    x1 = L2.head
    x2 = L2.tail
    L2.head = None
    L2.tail = None
    x1.prev = x
    x.next = x1
    x2.next = xn
    if xn:
        xn.prev = x2
    else:
        self.tail = x2