IE 172 Midterm Practice Problems Partial Solutions

Dr. T.K. Ralphs

March 23, 2015

1. **Consider the following algorithm for finding the minimum value in an `n` $\times$ `n` matrix `M` stored as a list of lists.**
   - If `n = 1`, then return `M[0][0]`
   - Otherwise, recursively find the minimum as follows.
     - Divide the matrix into four equal submatrices of (approximate) size `n/2` $\times$ `n/2` and recursively find the minimum of each submatrix.
     - Return the minimum of the mimima among the four submatrices as the overall minimum.

   (a) **Write a recurrence for the running time of this function in terms of `n`.**

   If we consider the running time as a function of `n`, then we have four recursive calls with input size equal to `n/2` for each call. Furthermore, we have a base case of `n = 1`. So the recurrence is

$$\begin{cases} T(1) = 1, & n = 1 \\ T(n) = 4n/2 + c, & \text{otherwise.} \end{cases}$$

   Here, c represents the number of steps that need to be done aside from the recursive calls (primarily combining the results of the recursive calls to obtain the overall minimum).

   (b) **Solve the recurrence and compare the running time with that of a straightforward method of finding the minimum simply by searching the elements of the matrix linearly.**

   We can solve this recurrence by the Master Theorem. We have $a = 4$ and $b = 2$, so that $n^{\log_b a} = n^2$. Since $f(n) = c$, we have that $f(n) \in o(n^{\log_b a})$ so we are in case 1. Thus, $T \in \theta(n^2)$. This is actually the same running time as the naive algorithm of searching through the matrix sequentially, since the number of entries in the matrix is also n^2. Thus, this recursive algorithm does not improve on the sequential algorithm.

2. **Consider the following recursive function for determining the lengths and positions of the hash marks to be placed on a ruler. Keep in mind what a standard 12 inch ruler typically looks like—the longest hash marks are drawn every inch, then shorter ones every half inch (unless there is already a longer mark there), then even shorter ones every quarter inch and so on. The code below is to figure out the location and length of the hash marks for a ruler of a given length.**

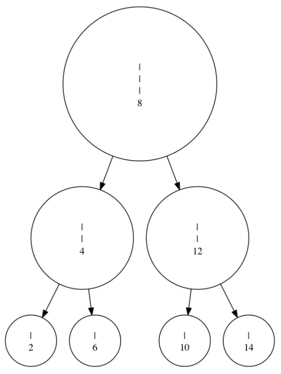

Figure 1: Visualization of the call tree for the `make_ruler()` function.

```
def ruler(hash_marks, left = 0, right = 16, length = 3):
    middle = (left+right)//2
    if length > 0:
        ruler(hash_marks, left, middle, length - 1)
        hash_marks[middle] = length
        ruler(hash_marks, middle, right, length - 1)
```

(a) **What is the output of the `make_ruler()` function below (you may not execute the code to find the answer)**?

```
def make_ruler():
    hm = {}
    ruler(hm)
    print hm
```

The output is

```
{2: 1, 4: 2, 6: 1, 8: 3, 10: 1, 12: 2, 14: 1}
```

This represents the locations and lengths of the hash marks on a 16-inch ruler. A visualization of the execution of the above recursive code in a call tree is shown in Figure **??**.

(b) **Write a recurrence for the running time of the function call `ruler(hm, l, r, h)` and solve it. (Hint: First determine what the running time depends on.)**

The running time of this function depends only on the initial value of `h`. In each recursive call, the value of this variable is reduced by 1 and the base case for the recursion (the leaf nodes of the above tree) is reached when the height is 1. Therefore, the running time recurrence is:

$$\begin{cases} T(\mathtt{h}) = 1, & \mathtt{h} = 1 \\ T(\mathtt{h}) = 2T(\mathtt{h}-1) + c, & \text{otherwise.} \end{cases}$$

By telescoping, the solution to this recurrence is

$$T(\mathtt{h}) = 1 + 2T(\mathtt{h}-1) = 1 + 2 + 2T(\mathtt{h}-2) = \sum_{i=0}^{\mathtt{h-1}} 2^i. \quad (1)$$

The solution to this equation is $2^{\mathtt{h}} - 1$.

(c) **Write a simple nonrecursive version of `ruler` that produces the same output when called from `make_ruler` (Hint: It can be done in three lines of code).**

Here is a simple non-recursive function that produces the same result as the recursive function.

```
def ruler_nr(hash_marks, left = 0, right = 16, height = 3):
    for i in range(height):
        for j in range(left+right//(2**(height-i)), right,
                       right//(2**(height-i))):
            hash_marks[j] = i+1
```

The reason this works is because we first assign a hash mark of height 1 to all possible ruler locations. We then assign a height of 2 to every other location (which over-writes the original assignment of height 1). This continues until the final iteration when the single location overwritten is middle of the entire range, which gets the biggest hash mark.

3. **Sorting of a linked list can be challenging, since many of the basic operations one can do on lists are inefficient with linked lists. For example, we cannot swap items `i` and `j` stored in a linked list in constant time as we can with a Python list. This makes methods like quick sort inefficient if the list is stored in a linked list. There are two ways around this. The first is to copy the list from a linked list into an array and then do a normal sort (copying it back into the linked list again if necessary afterwards. The other is to use a sorting method that works well with linked lists.**

   (a) **Write a Python function for copying a linked list into a standard Python list and say what it's running time is. You can write your function in terms of the high-level API for the `LinkedList` class we used in the lab or just directly using the methods of the `Node` class. Don't worry about getting the names of the functions exactly right, as long as the idea is clear. Would copying the linked list into a Python list and then sorting it there be an efficient method of sorting the linked list?**

   A straightforward way to do this would be

```
def copy_linked_list(llist):
    plist = []
    for i in llist:
        plist.append(i)
    return plist
```

   Technically, this has a running time of n^2, however, since linked lists are stored with the last element of the list at the head, which means that reverse iteration is linear time, whereas forward iteration is quadratic. A better implementation would be

```
def copy_linked_list(llist):
    plist = []
    for i in reversed(llist):
        plist.append(i)
    return reversed(plist)
```

Of course, we don't really need to reverse `plist` if we are just going to sort it later anyway. Since copying the list is a linear time operation, doing a copy first and then sorting the list would be a reasonable approach. It would not change the worst case running time in principle, but one would have to check what the empirical performance of this approach is.

(b) **Using merge sort is a reasonable alternative. Merging of sorted linked lists can be just as efficient as merging of sorted Python lists and it can be done without allocating any additional memory. Write a Python function to merge two sorted linked lists. In this case, you will probably need to use the API of the node class directly.**

Assuming the lists are sorted initially so that the smallest element is at the head of each linked list (from largest to smallest with respect to the standard way of indexing elements of the list), then the following function does the merge.

```
def merge(head1, head2):
    i = head1
    j = head2
    if i.getData() <= j.getData():
        head = i
        current = i
        i = i.getNext()
    else:
        head = j
        current = j
        j = j.getNext()
    while i != None and j != None:
        if i.getData() <= j.getData():
            current.setNext(i)
            current = i
            i = i.getNext()
        else:
            current.setNext(j)
            current = j
            j = j.getNext()
    if i == None:
        currrent.setNext(j)
        return head
    if j == None:
        currrent.setNext(i)
        return head
```

(c) **Describe how to use the merge operation from the previous part to implement a merge sort algorithm that directly sorts a linked list in $n \lg n$ time without having to allocate any additional space. You can do with this with code (perhaps the easiest way) or describe the approach in words. (Hint: How would you implement the recursive function itself once you have the merge function? Write the code for this and then explain why it's running time is the same as the merge sort routine we discussed in class).**

With the `merge` operation implemented, we merely need to determine how to split the list. The same basic recursive method used with a standard list works. We only need a slight modification.

```
def merge_sort(head, length):
    if length == 1:
        return
    head1 = head
    for i in range(length/2):
        head = head.getNext()
    head2 = head.getNext()
    head.setNext(None)
    head1 = merge_sort(head1, length/2)
    head2 = merge_sort(head2, length - length/2)
    return merge(head1, head2)
```

4. **The following questions deal with recurrences. In all cases, assume $T(1) = 1$.**

   (a) **Solve the recurrence $T(n) = (T(\frac{n}{2}))^2$.**

   To solve this recurrence, we can use telescoping in the usual way.

$$\begin{aligned} T(n) &= T((\frac{n}{2}))^2 && (2) \\ &= T((\frac{n}{4}))^4 && (3) \\ &= T((\frac{n}{2^k}))^{2^k},\ k \leq \lceil \lg n \rceil. && (4) \end{aligned}$$

   Substituting $k = \lceil \lg n \rceil$, we get $T(n) \in \theta(1)$.

   (b) **Solve the recurrence $T(n) = 9T(n/3) + n^2$**

   This recurrence is in a form such that we can apply the Master Theorem. We have $a = 9$ and $b = 3$, which means that $\log_b a = 2$ and we have that $f(n) = n^{\log_b a} = n^2$, so we are in case (ii). We thus have $T \in \Theta(n^2 \lg n)$.

   (c) **Solve the recurrence $T(n) = 2T(n/2) + \sqrt{n}$.**

   This recurrence is also in a form such that we can apply the Master Theorem. We have $a = 2$ and $b = 2$, which means that $\log_b a = 1$ and we have that $\sqrt{n} \in o(n^{\log_b a})$, so we are in case (i). We thus have $T \in \Theta(n)$.

   (d) **Determine the solution to the recurrence**

$$T(n) = \alpha T(n/2) + n^2.$$

   for all values of $\alpha \geq 1$. In other words, state all solutions to this recurrence for different values of $\alpha \geq 1$ and state for which values of α each solution would arise.

   Here, we can use the Master Theorem. There will be at most three different solutions, depending on which of the three cases we are in. We have to determine for which values of α each case will occur. To determine this, we compare the polynomials $n^{\lg \alpha}$ and n^2, which is equivalent to comparing $\lg \alpha$ to 2. The three cases are:

- $\underline{\alpha > 4}$: This occurs when $\lg \alpha > 2$ and puts us in the first case for the Master Theorem, which means that $T(n) \in \theta(n^{\lg \alpha})$.
- $\underline{\alpha = 4}$: This occurs when $\lg \alpha = 2$ and puts us in the second case for the Master Theorem, which means that $T(n) \in \theta(n^2 \lg n)$.
- $\underline{\alpha < 4}$: This occurs when $\lg \alpha < 2$ and puts us in the third case for the Master Theorem, which means that $T(n) \in \theta(n^2)$. For this case, we must also verify that there is a constant $c < 1$ such that $\alpha \left(\frac{n}{2}\right)^2 \leq cn^2$. Taking $c = \alpha/4$ shows this condition is satisfied.

(e) **Solve the recurrence** $T(n) = 3T(n/9) + n^2$: To solve this recurrence, we can apply the Master Theorem. We have $a = 3$, $b = 9$, and $f(n) = n^2$. Thus, we have $\log_b a = 1/2$ and $f(n) \in \Omega(n^{\log_b a})$. We conclude that $f \in \Theta(n^2)$.

(f) **Solve the recurrence** $T(n) = T(n-1) + n^2$: In this case, we use telescoping. We have

$$\begin{aligned} T(n) &= T(n-1) + n^2 && (5)\\ &= T(n-2) + (n-1)^2 + n^2 && (6)\\ &= \sum_{i=0}^{n-1} (n-i)^2 && (7)\\ &= \sum_{i=1}^{n} i^2 && (8)\\ &= n^3/3 + n^2/2 + n/6 && (9)\\ &\in \Theta(n^3) && (10) \end{aligned}$$

(g) **Solve the recurrence** $T(n) = T(n-2) + n/2$**.**

We use telescoping to solve this recurrence. We have

$$\begin{aligned} T(n) &= n/2 + T(n-2) && (11)\\ &= n/2 + (n/2 - 1) + T(n-4) && (12)\\ &= n/2 + (n/2 - 1) + (n/2 - 2) + T(n-4) && (13)\\ &= \sum_{i=1}^{n/2} i && (14)\\ &= \frac{n/2(n/2-1)}{2}. && (15) \end{aligned}$$

(h) **Solve the recurrence** $T(n) = 7T(n/3) + n^2$**.**

This one can be solved using the Master Theorem. We have $a = 7$, $b = 3$, and hence $\log_b a = 1.77$. Examining the cases, we are in case 3, since

$$f(n) = n^2 = n^{\log_b a + \epsilon} \quad (16)$$

where $\epsilon = .23$. We need to also check that the second condition holds, that is $af(n/b) = 7f(n/3) = 7n^2/9 \leq cn^2$ for some constant $c < 1$. This is certainly true by taking $c = 7/9$. Hence, we have $T(n) \in \Theta(n^2)$.

5. **Consider the following Python function**:

```
def foo(mystring, beg = None, end = None):
    if  beg == None:
        beg = 0
    if end == None:
        end = len(mystring)
    if beg == end:
        if mystring[beg] in '1234567890':
            return mystring[beg]
        else:
            return ''
    else:
        return (foo(mystring, beg, (beg + end)/2) +
                foo(mystring, (beg + end)/2 + 1, end))
```

(a) **What does this function do? Write a documentation string for the function, being sure to specify any pre-conditions required in order for the function to run properly.**

This function outputs a string containing all digits 1 through 9 that occur in the string `mystring[beg:end+1]` in the same order and duplicity as they appear in the string. In other words, it strips all characters except for the digits 1 through 9 from the string. Here is a documentation string.

```
'''This function outputs a string containing all digits 1 through 9 that occur
   in the string 'mystring[beg:end+1]' in the same order and duplicity
   as they appear in 'mystring'.

   pre: 0 <= beg <= end <= len(mystring) - 1
   post: The output string described above is returned
'''
```

(b) **What is the worst-case (theoretical) running time of this function? Make sure you consider all the components of the algorithm and explain your analysis completely.**

This function is recursive, so we can write the recurrence and solve it. The running time depends of the values of `beg` and `end`, which determine the effective length of the input (the part of the whole string that is actually operated on during the function call). The way the function is written, the initial call could just be `foo(mystring)`, since the default values of `beg` and `end` are set automatically when the function is called by a user from outside the function itself. This ensures that the entire string is considered in the outermost call. The length of the original input string after this first call is then `n = end-beg +1`.

Each recursive call works on just a part of the original string and the part it works on is determined by `beg` and `end`. If `beg = 3` and `end = 5`, for example, the function will only pay attention to the substring including characters 3 to 5, i.e., `mystring[3:6]`. If the original string were `mystring = 'My number is 5551212'`, then the substring `mystring[3:6]` would be `'num'`. This means that the size of the input for each function call (the size of the substring on which each function call operates) is just `end-beg+1`.

To write the recurrence, we need to figure out how much of the original input is passed to each of the recursive calls (and how many recursive calls there are). The original input size is beg-end+1 and the two recursive calls have inputs of sizes `beg, (beg+end)/2` and `(beg+end)/2 + 1, end`. In other words, they each get half of the original list, e.g., `(beg+end)/2 - beg = (end - beg)/2`. If each recursive call gets half of the original list and there are two recursive calls, then the recurrence is $T(n) = 2T(n/2)+1$. This has the right form to apply the Master Theorem and we have that $a = b = 2$ and $f(n) = 1$. Applying the Master Theorem, we are in case 1, since $n^{\log_b a} = n$. The running time is thus just n.

6. **Consider the following Python function**

```
def foo(A, B):
    m = len(A)
    n = len(B)
    if n != len(A[0]):
        print 'Dimensions don't match! Exiting...'
        return None
    p = len(B[0])

    C = [[] for i in range(m)]
    for i in range(m):
        for k in range(p):
            x = 0
            for j in range(n):
                x += A[i][j] * B[j][k]
            C[i].append(x)
    return C
```

(a) **What does this function do?**

This function multiplies two matrices stored in `A` and `B`. When you multiply two matrices, they need to have matching dimensions. In this case, we have that `A` is an `m` $\times$ `n` matrix and `B` is an `n` $\times$ `p` matrix. The result is an `m` $\times$ `p` matrix. The function checks to make sure the dimensions match and returns `None` otherwise.

(b) **What is the running time of this function in terms of `m`, `n`, and `p`?**

The running time is $\Theta(mnp)$ because of the triple loop structure.

(c) **Write a documentation string for the function, being sure to specify any pre-conditions required in order for the function to run properly.**

Here is a documentation string.

```
'''
This function multiplies two matrices.

pre: A and B are each a list of lists representing the two two
matrices. Thus, the lists stored in A should all have the same
```

```
length and should contain objects for which the multiplication
and addition operator is defined. The length of each the lists
in A should be equal to the length of B.

post: The return value is a list of lists containing the matrix
product of the input matrices A and B. Thus the length of C is
equal to the length of A and the length of each of the lists in
C is equal to the length of each of the lists in B.
'''
```

7. (a) **Consider the following simple Python function.**

```
def bar(i):
    return i & 1
```

Say concisely what this function does.

The `&` symbol is a bitwise AND operator. When you take `i & j`, the numbers `i` and `j` first get represented as binary numbers with enough leading zeros on the smaller one to make them have the same number of binary digits. The result of the operation is another binary number that has a 1 wherever both `i` and `j` have a 1 and zeros elsewhere. In this case, since we are taking `i & 1`, the second number only has a one in the last binary digit and zeros everywhere else. This means the result will be 1 if `i` has a one in its last digit and 0 otherwise. When you represent a number in binary, the last digit tells you whether the number is even or odd. So this function just tests to see whether `i` is even or odd.

(b) **The function `what` below uses the function `bar` above as a subroutine.**

```
def what(aList):
    i = 0
    j = len(aList)
    while True:
        while i <= last and bar(aList[i]):
            i += 1
        while j >= 0 and not bar(aList[j]):
            j -= 1
        if i >= j:
            break
        else:
            aList[i], aList[j] = aList[j], aList[i]
    if bar(aList[j]):
        return j - 1
    else:
        return j
```

What does this function do?

This function rearranges the list so that all the odd numbers come first and the even numbers come second.

8. (a) **Write the Python code for a modified version of the quicksort algorithm that simply finds the *median* of a list of numbers rather than sorting the**

whole list. Just write the main recursion—you can assume you already have a working `partition` function.

```
def find_median_r(aList, first, last):
    if last > first:
        midpoint = len(aList)/2
        pivot = partition(aList, first, last)
        if pivot == midpoint:
            return aList[pivot]
        elif pivot > midpoint:
            return find_median_r(aList, first, pivot - 1)
        elif pivot < midpoint:
            return find_median_r(aList, pivot + 1, last)
```

(b) **Write the recurrence for the worst-case running time of your algorithm and solve it.**

The worst case here is similar to quick sort. If (by chance) we choose the pivot element to be the largest (or smallest) element in the array every time, then we end up with one recursive call on a list of size $n-1$ and the other on a list of size 1. Taking into account that the call on the list of size 1 takes constant time and that the partitioning takes linear time, the recurrence is

$$T(n) = T(n-1) + n \tag{17}$$

Note, however, that the algorithm terminates when the pivot element is the middle element of the list (when the input size to the recursion is $n/2$), so even in this worst case, we don't have to execute quite as many recursive calls as in quick sort. The solution to this is

$$T(n) = \sum_{i=n/2}^{n} i \in O(n^2) \tag{18}$$

(c) **Briefly describe another algorithm with a better worst-case running time and say what its running time is.**

The simplest algorithm is just to sort the array and then pick the middle element. With this algorithm, we can achieve a worst-case running time of $O(n \lg n)$, which is better than the recursive algorithm above in the worst case. This is exactly the same as the situation with quick sort itself—quick sort has a worst-case running time of n^2, but an optimal sorting algorithm would have a worst-case running time of $n \lg n$.

(d) **We discussed in class that the worst-case behavior of quicksort comes from making a bad choice of pivot element. The best case would be if we could somehow choose the pivot element to be the median every time. Suppose you use your algorithm for finding the median in order to choose the ideal pivot element for quicksort. Is this a good idea? What would the running time of the modified quicksort algorithm be in the worst case? Write the recurrence and solve it.**

If we use an $O(n \lg n)$ algorithm for finding the median each time, then the quick sort recurrence becomes

$$T(n) = 2T(n/2) + n \lg n \tag{19}$$

By the Master Theorem, we can easily derive that $T(n) \in \Theta(n \lg n)$ ($a = b = 2$ and we are in the third case, since $n \lg n \in \Omega(n)$.

9. **Consider the following function**:

```
def foobar(n):
   for i in range(n):
      for j in range(n):
         for k  range(j:n):
            print '(', i, ', ', j, ', ', k, ')'
```

   (a) **State succinctly what this function does**.

   This function prints out all triples (i, j, k) of integers between 0 and $n-1$ such that $j \leq k$.

   (b) **What is the exact number of lines of text it outputs as a function of n?**

   Each outer loop iteration prints out exactly the same number of lines of text, since the inner loops don't depend on the value of i. Hence, we just need to know the number of lines printed by the two inner loops. The overall total will be this quantity multiplied by n. The number of lines of output printed by the two inner loops is

$$\sum_{l=0}^{n-1} l = n(n+1)/2. \tag{20}$$

   Therefore, the overall total number of lines printed for each call to `foobar(n)` is $n^2(n+1)/2$.

   (c) **If the actual CPU time required to execute `foobar(10)` is 1 second, what is the predicted CPU time in seconds for execution as a function of n?**

   As a simple model, we will assume that the running time is proportional to the number of lines of text printed. Hence, based on the answer to part (b), we assume that the running time is

$$T(n) = cn^2(n+1)/2 \tag{21}$$

   for some unknown constant c. Since the running time for `foobar(10)` is 1 second, we conclude $c = 1/550$, so that

$$T(n) = n^2(n+1)/1100. \tag{22}$$

10. You are given information that the running time of algorithm A is $O(n \lg n)$ and that the running time of algorithm B is $O(n^3)$. What does this tell you about the relative performance of A and B?

    For large enough input sizes, algorithm A will outperform algorithm B. For small input sizes, you cannot say definitively, but given the difference in theoretical running time, it is still likely that algorithm A will be better.

11. Show that $n \lg n \in O(n^{3/2})$. Is it also true that $n \lg n \in o(n^{3/2})$?

Applying L'Hospital's rules several times, we have

$$\begin{aligned}
\lim_{n\to\infty}\frac{n\lg n}{n^{3/2}} &= \lim_{n\to\infty}\frac{\lg n+1}{3\sqrt{n}/2} && (23)\\
&= \lim_{n\to\infty}\frac{4\sqrt{n}}{3n} && (24)\\
&= \lim_{n\to\infty}\frac{2}{3\sqrt{n}} && (25)\\
&= 0 && (26)
\end{aligned}$$

Hence, $n\lg n \in o(n^{3/2})$

12. **For each pair of functions f and g, state whether f is o, O, Θ, Ω, ω of g (choose as many as apply). Prove your assertion. Choose 2 of these 3 problems.**

    (a) $f(n) = \lg n$, $g(n) = \ln n$.
    In this case, we have $f = \frac{1}{\ln 2}g$, which immediately implies that $f \in \Theta(g)$. This in turn implies that $f \in O(g)$ and $f \in \Omega(g)$.

    (b) $f(n) = n^{1/n}$, $g(n) = 1$.
    First, we show that $\lim_{n\to\infty} f(n) = 1$. The easiest way to see this is to observe that $\lim_{n\to\infty} \lg f(n) = (\lg n)/n = 0$. Then, since $f = f/g$, we have $\lim_{n\to\infty} f(n)/g(n) = 1$ and hence $f \in \Theta(g)$. This further implies that $f \in O(g)$ and $f \in \Omega(g)$.

    (c) $f(n) = n^{\lg n}$, $g(n) = n$.
    We have $\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} n^{\lg n-1} = \infty$. Hence, $f \in \omega(g)$ and also $f \in \Omega(g)$.

13. **A *generalized queue* or a *deque* is a data structure that combines the functionality of a stack, in which the item most recently added to the list is the next to be removed (LIFO), with that of a standard (FIFO) queue, in which the item least recently added to the list is the next to be removed. In a basic generalized queue interface, the following operations are supported.**

    - `addFront`**: add an item to the front of the queue.**
    - `addRear`**: add an item to the rear of the queue.**
    - `removeFront`**: return a copy of the item at the front of the queue and then delete it from the queue.**
    - `removeRear`**: return a copy of the item at the ear of the queue and then delete it from the queue.**
    - `isEmpty`**: indicate whether the queue is empty or not.**
    - `size`**: indicate whether the queue is empty or not.**

    Note that we don't have operations to look at items in the queue without deleting them.

    (a) **What data attributes would be needed to implement a generalized queue in order to allow all operations above to be performed in constant time (consider how you are going to implement the operations)?**
    There are a number of ways of doing this and each has different efficiency for different operations. The simplest of these would be to implement this data structure on top of a Python list. We need the following data attributes:

- `aList` (the list itself),
- `front` (index of the most recent element inserted),
- `back` (index of the least recent element inserted), and
- `empty` (a Boolean that indicates whether the queue is empty).

When an item is inserted, it is simply appended to the list (placed in slot `(self.front + 1)`). When removing the most recent item, we can either use the list's `pop()` method or decrement `front`. Using the latter method, we would need to keep track of the size of the list to determine whether to use append or assignment for future additions to the list. When removing the least recent item, we just have to increment `self.back`.

Note that with this approach, the amount of memory allocated for the underlying list can grow, even if the number of items in the queue does not, especially if the deque is used as a regular queue (each time the least recent item is removed and another item replaces it, the size of the underlying list increases by 1. To combat this, we can fix a maximum size for the queue (and the underlying list) by "wrapping around" when we reach the end of the list (place new items in slot `(self.front + 1) % self.maxSize`, provided that slot is empty. If the slot is not empty, the queue is full, causing an overflow. The reason for using modular arithmetic here is to wrap-around to the beginning of the array when reaching the end. When deleting, we simply update the appropriate pointer (also using modular arithmetic).

(b) **Write a constant-time `addFront` operation.**

For the implementation with fixed list size, we would have

```
def addFront(self, items)
    if (self.front + 1) % self.maxSize) == self.back):
        raise Exception, "List overflow"
    else:
        self.front = (self.front + 1) % self.maxSize
        self.aList[self.front] = item
```

(c) **Write pseudo-code for a constant-time `removeFront` operation.**

```
def removeFront(self):
    if self.empty:
        raise Exception, "List is empty"
    if front == back:
        self.empty = True
        return self.aList[self.front]
    self.front = (self.front - 1) % self.maxSize
    return self.aList[self.front + 1]
```

14. Suppose you are performing double hashing and you have a bug in your hash function code so that it always returns the same value. What happens when (i) the first hash value is wrong, (ii) the second hash value is wrong, (iii) both hash values are wrong.

If you always get the same first hash value, the hash table will perform correctly, just not very efficiently, since every item inserted after the first one will be inserted into a slot already occupied. With the second hash value wrong, the table will perform correctly as long as the second hash value is relatively prime to the table size. The performance will likely be similar to linear probing, since all items will get the same increment in searching for empty slots in the table. If both hash values are wrong, everything will still work correctly, as long as the second hash value is relatively prime to the table size, but inserting and searching will have running times that are linear in the size of the table. This would be the same as just putting the items into a regular list and searching sequentially.

15. (a) **State an algorithm for determining whether two strings are *anagrams*, i.e., have the same set of characters but in different orders. There are at least two "obvious" solutions for this—try to choose the one that is more efficient.**

    There are several ways of doing this. The most naive way is with a double loop. We simply consider each character of the first string in turn and look for it within the second string. A better algorithm is to sort the letters in each of the two strings first and then check the two strings for equality. There are two different sorting algorithms we could use. First, we could use a standard comparison-based sort, such as merge sort or quicksort. Assuming we know the set of characters that could possibly occur in the string and that this set is relatively small, we could also use a bucket sort, which would consist of counting how many times each character occurs in each of the strings and then comparing the counts. If the counts are the same for every character, then the sorted strings are the same.

    (b) **Determine the worst-case theoretical running time of your algorithm. Explain your reasoning.**

    The running time of the naive algorithm is $O(mn)$, where m and n are the lengths of the two strings. The running time of the algorithm utilizing comparison-based sorting would be $O(k \log k)$, where $k = \max\{m, n\}$. The running time of the bucket sort variant would be $O(n + M)$. Technically, the bucket sort would be a linear time algorithm and hence the most efficient from an asymptotic viewpoint. However, if the length of the strings is much smaller than M, which they typically would be, then the method based on comparison-based sorting would be better.

16. `Python has a Set data structure that implements the operations one would typically perform on a mathematical set. Three ways of implementing such a data structure would be to store the items in (1) a Python dictionary, (2) a Python list, or (3) an OrderedList (like the one from Lab 2). For each method below, state which way of storing the set would result in the most efficient implementation. Explain your reasoning thoroughly!`

    (a) `add()`: For adding, the Python list would be constant time in the worst case; the dictionary would be constant time in the average case, assuming a fixed load factor, and linear time in the worst case; while the ordered list would be linear time in the worst case.

    (b) `remove()`: For removal, the Python list would require linear time in the worst case, since we have to search for the item; the dictionary would again be constant time in the average case, assuming a fixed load factor, and linear time in the worst case (if we use sentinels for deletion); the ordered list would require log time to find the item (assuming a Python list and not a linked list), but actual removal would still require linear time, since we would need to shift the items to fill the gap left by the removed item (unless we used sentinels in this case as well).

    (c) `__contains__()`: The analysis would be the same as for `remove()` except that since we only have to locate the item and not actually delete it, the ordered list would only require log time (again assuming the ordered list is stored in a Python list and not a linked list).

(d) `intersection()`: The simplest way to do intersection would be to iterate over the elements of one set and call the `__contains__` of the second set with respect to each one of them. With this implementation, the running time would vary in a fashion similar to that of the `__contains__` method. In the case of either the Python list or the dictionary, one could sort the two lists first, allowing for a more efficient determination of the intersection. This is similar to the algorithm for the finding of anagrams using comparison-based sorting in the question above.

(e) `union()`: The answer to this question depends on whether one allows for duplicated items in the set. Typically, a set data structure would not allow for duplicated items (it's about membership, not cardinality), but for efficiency reason, this requirement could be relaxed. With a Python list, if one is not worried about duplication, then one would simply concatenate the two lists. For both the dictionary implementations, it would be difficult to avoid inserting the elements of one set into the other one by one. It's possible one could avoid this in the case of a dictionary by using chaining with the same hash function and table size, in which case the lists associated with each table slot could be concatenated and sorting could be avoided. For the ordered list implementation, we can merge the two lists just as do in merge sort. If we want to avoid duplication, then this last one is the most efficient method of doing union.