IE 172 Midterm Practice Problems

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`.

   (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.

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.

```
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
```

   (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).

(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).

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?

   (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.

   (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).

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

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

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

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

   (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. Justify your answer.

   (e) Solve the recurrence $T(n) = 3T(n/9) + n^2$.

   (f) Solve the recurrence $T(n) = T(n-1) + n^2$.

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

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

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) (10 points) 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.

(b) (10 points) 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.

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) (10 points) What does this function do?

(b) (10 points) What is the running time of this function in terms of `m`, `n`, and `p`?

(c) (10 points) Write a documentation string for the function, being sure to specify any pre-conditions required in order for the function to run properly.

7. (a) (10 points) Consider the following simple Python function.

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

Say concisely what this function does.

(b) (10 points) 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?

8. (a) (10 points) 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.

   (b) (10 points) Write the recurrence for the worst-case running time of your algorithm and solve it.

   (c) (10 points) Briefly describe another algorithm with a better worst-case running time and say what its running time is.

   (d) (10 points) 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 quicksort algorithm be in the worst case if we choose the pivot to be the median each time? Write the recurrence and solve it.

   (e) (10 points) Write the Python code for an algorithm similar to quicksort that finds the *median* of a list of numbers but without sorting the whole list. Just write the main recursion—you can assume you already have a working `partition` function from a regular quicksort implementation.

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.

(b) What is the exact number of lines of text it outputs as a function of n? Show your work.

(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? Show your work.

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?

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

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.

    (a) $f(n) = \lg n$, $g(n) = \ln n$.

    (b) $f(n) = n^{1/n}$, $g(n) = 1$.

    (c) $f(n) = n^{\lg n}$, $g(n) = n$.

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)?

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

    (c) Write a constant-time `removeFront` operation.

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.

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.

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

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()`

    (b) `remove()`

    (c) `__contains__()`

    (d) `intersection()`

    (e) `union()`

17. Consider the following algorithm for finding the median of an array of n integers. You may assume that the values in the array are distinct and that n is a power of 5.

    - Arbitrarily insert the elements of the array into a $5 \times n/5$ matrix.
    - Sort each column of the matrix.
    - Recursively find the median of the middle row of the matrix.

    (a) Is this algorithm correct?

    (b) Write a recurrence for the worst-case running time of this algorithm and solve it. Justify all steps of your answer.

    (c) Describe a version of the algorithm in which the array is first arranged into an $k \times n/k$ matrix and discuss how the worst-case running time changes.

18. The following questions refer to the mergesort algorithm. Assume that we are sorting from smallest to largest.

    (a) Consider an implementation of mergesort that divides the array into k approximately equal parts, where k is a constant, instead of the usual two. Determine the worst-case running time of this algorithm if implemented using a straightforward recursion. Justify all your steps (Hint: Set up a recurrence and solve it using standard techniques.)

    (b) The classical implementation of mergesort is considered *non-adaptive*, since the running time does not depend on the input array. In other words, every input array requires $\Theta(n \lg n)$ steps to sort, regardless of its order before sorting. Mergesort can be made partially adaptive by checking in the merge step whether the largest element of the first subarray to be merged is smaller than the smallest element of the second subarray. Use this idea to explain how to improve the best-case running time. What is the new best case and on what type of arrays is the best case achieved? Justify your answer.

    (c) Consider a version of mergesort in which the array is divided into two pieces of random size, then recursively sorted as usual, and merged to form a completely sorted array. What is the worst-case running time of this algorithm? Justify your answer.

19. The following questions refer to the open addressing implementation of the bash table class from Laboratory 6.

    (a) Below is the insertion function for an open addressing implementation of a hash table similar to what you were given in Laboratory 6 except that this implementation uses linear probing.

```
def _lookup(self, key):
    """
    Find the entry for a key.
    """
    key_hash = self.first_hash(key)
    entry = self.table[key_hash]
    if entry.key is None or entry is key:
        return entry
    free = None
    if entry.key is dummy:
        free = entry
    elif entry.hash == key_hash and key == entry.key:
        return entry

    i = key_hash
    while True:
        i += 1
        i = i % self.size
        entry = self.table[i]
        if entry.key is None:
            return entry if free is None else free
        if (entry.key is key or
            (entry.hash == key_hash and key == entry.key)):
            return entry
        elif entry.key is dummy and free is None:
            free = dummy
```

    Explain how this function would need to be modified in order to implement double hashing. Rewrite the exact lines of this function that need to be modified and also discuss any changes that would need to be made to other functions or classes to support double hashing.

    (b) In double hashing, explain what would happen if the same function were used for both the first and second hash functions.

20. Analyze the running time of the following function

```
def weird(n):
   for i in range(n*log(n)):
      j = i
      while j <= n:
         j += 1
```

21. Is it always the case that $f(2n) \in O(f(n))$? Prove or provide a counterexample.

22. One difficulty in working with functions involving $n!$ is that it cannot actually be computed for large n without causing an overflow. Describe a method for computing the quantity ($n!$ mod M) such that overflow is not an issue.

23. The following questions refer to a generic list data structure. Recall that a list can be stored either as an array or a linked list.

    (a) Write a function that moves the smallest item to the beginning of the list. What is the running time of this operation in the worst case for both a Python list and a linked list?

    (b) Describe how to use the function from parts (a) and (b) to sort an array. What is the worst-case running time of this sorting algorithm? What sorting algorithm is this equivalent to?

24. A *Josephus election* is conducted as follows. The n candidates sit in a circle. Starting with a randomly chosen spot in the circle, the candidates count off and every m^{th} candidate is eliminated until only one remains.

    (a) Describe how to implement an algorithm for determining the winner of a Josephus election using modular arithmetic.

    (b) What is the running time of your algorithm, in terms of m and n.

    (c) How much memory does the algorithm use?

25. In this problem, we consider a data structure for storing a document in a simple text editor. The goal of our data structure is to allow basic editing operations, such moving the cursor and entering text, to be performed efficiently. We consider here a keyboard-based system, so the only way to move around the document is to use the arrow keys.

    The data structure that has been proposed is to store each line of the document as a linked list and to store the entire document as a (regular Python) list of these linked lists.

    (a) What is the running time of the following basic operations?

    - Typing a character
    - Deleting a character
    - Hitting one of the arrow keys (up, down, right, left).

    You can consider the input to be a function of the maximum number of characters per line C and the number of lines N. Note that without some kind of line wrapping capability, the size of a single line can grow quite large. In your analysis, consider a carriage return (new line—what you get when hit the `Return` key) to be a character (you might want to treat this as a special case).

    (b) Suppose we want to include a "word wrap" capability. In other words, each time the typing of character causes a line to exceed a certain maximum length, the last word on the offending line must be moved to the next line. Write a function to insert a character into a line, taking into account word wrap, keeping in mind that the wrapping of one line may necessitate the wrapping of additional lines.

    (c) Suggest an algorithm for efficiently spell-checking the document and analyze its runing time. There are multiple ways of doing this. Try to choose the most efficient one.