Quiz 1 Solutions

Problem 1. Asymptotic growth [10 points]

For each pair of functions $f(n)$ and $g(n)$ given below:

- Write Θ in the box if $f(n) = \Theta(g(n))$
- Write O in the box if $f(n) = O(g(n))$
- Write Ω in the box if $f(n) = \Omega(g(n))$
- Write X in the box if none of these relations holds

If more than one such relation holds, write only the strongest one. No explanation needed. No partial credit.

O, Θ, Ω or X	$f(n)$	$g(n)$
O	n^2	n^3
Ω	$n \lg n$	n
Θ	1	$2 + \sin n$
Ω	3^n	2^n
Θ	4^{n+4}	2^{2n+2}
O	$n \lg n$	$n^{101/100}$
Θ	$\lg \sqrt{10n}$	$\lg n^3$
O	$n!$	$(n+1)!$

Problem 2. Miscellaneous True/False [15 points] (5 parts)

For each of the following questions, circle either T (True) or F (False). **Explain your choice.** (No credit if no explanation given.)

(a) T F A hash table guarantees constant lookup time.
Explain:

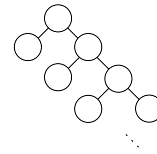
Solution: False. It only has *expected* constant lookup time; if $\Theta(n)$ elements collide, then lookup may take $\Theta(n)$ time in the worst case (assuming chaining).

(b) T F A non-uniform hash function is expected to produce worse performance for a hash table than a uniform hash function.
Explain:

Solution: True. A non-uniform hash function is more likely to result in collisions, which leads to slower lookup times.

(c) T F If every node in a binary tree has either 0 or 2 children, then the height of the tree is $\Theta(\lg n)$.
Explain:

Solution: False. One counterexample is a tree like the one shown here, extending down and to the right. It has $\Theta(n)$ height.



(d) T F A heap A has each key randomly increased or decreased by 1. The random choices are independent. We can restore the heap property on A in linear time.
Explain:

Solution: True. Simply call BUILD-HEAP at the root, which runs in $\Theta(n)$ time.

(e) T F An AVL tree is balanced, therefore a median of all elements in the tree is always at the root or one of its two children.
Explain:

Solution: False. An AVL tree doesn't guarantee that the left and right subtrees will be equal sizes; it only guarantees that the *heights* of the trees are close.

Problem 3. Sorting short answer [10 points] (3 parts)

(a) What is the worst-case running time of insertion sort? How would you order the elements in the input array to achieve the worst case?

Solution: Reverse order. $\Theta(n^2)$ running time.

(b) Name a sorting algorithm that operates in-place and in $\Theta(n \log n)$ time.

Solution: Heapsort.

(c) Write down the recurrence relation for the running time of merge sort. (You don't need to solve it.)

Solution: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$

Problem 4. Hashing [10 points] (4 parts)

Give a hash table that uses chaining to handle collisions, how would using sorted python lists in place of unsorted chains affect the following run times? Explain the circumstances of each of the four cases and **justify your choice**.

Solution notes:

- The question asks about using sorted python lists (a.k.a. arrays) instead of unsorted chains in each slot. The intent was *not* to ask about hash tables versus a single sorted list. The intent was that unsorted chains meant a linked list rather than a python list (a.k.a. array), but some students interpreted unsorted chain to mean unsorted python array, and we accepted that.
- As announced at quiz, the intent is to compare the best case of one versus the best case of the other, etc. (So they may be on different inputs.)

(a) Inserting an element (best case) using unsorted chains is
Slower / Neither Slower Nor Faster / Faster than using sorted python lists.

Solution: Neither. The best case here is when the slot the new element is going into is empty, so the running time is $\Theta(1)$ in both cases.

(b) Inserting an element (worst case) using unsorted chains is
Slower / Neither Slower Nor Faster / Faster than a sorted python lists.

Solution: Faster. In the worst case, all n previously hash elements went to the same slot, and the new element is going there as well. The sorted list takes time $\Theta(n)$ to insert the new element (since we may need to move all of the previously inserted elements within the array). The unsorted chain (assuming a linked-list implementation), only requires constant time.

(c) Finding an element (best case) using unsorted chains is
Slower / Neither Slower Nor Faster / Faster than using sorted python lists.

Solution: Neither. If the slot has only a single element, then the find requires time $\Theta(1)$ in either implementation.

(d) Finding an element (worst case) using unsorted chains is
Slower / Neither Slower Nor Faster / Faster than using sorted python lists.

Solution: Slower. If the slot has n elements, then finding one of them in an unsorted chain requires $\Theta(n)$ time in the worst case, while this requires only time $\Theta(\log(n))$ if you use binary search on a sorted array.

Problem 5. Sporadically-Rebalanced Trees [25 points] (3 parts)

Ben Bitdiddle has invented a new kind of data structure, which he calls a sporadically-rebalanced tree (SRT). Ben's tree is a binary search tree, with a twist: every time the size of the SRT doubles, it calls the REBALANCE procedure. That is, REBALANCE is called every time the SRT contains $n = 2^k$ nodes, where k is a natural number. REBALANCE rebalances the tree such that the height of an n element tree is $\Theta(\log n)$.

(a) Does Ben's scheme preserve the $\Theta(\log n)$ height of an n element tree? If so, explain why. If not, what is the worst-case height of an n element tree, in Θ notation?

Solution: No, worst case height is $\Theta(n)$. This happens when the insertions after a rebalancing operation add values larger than those in the tree, in increasing order, for example. This adds a long chain of nodes with one child to one leaf of the balanced tree.

(b) Argue briefly that rebalancing an n-node SRT can be done in $\Theta(n)$ time.

Solution: This was hard and very few students got it right. The trick is to almost start from scratch, rather than try to rebalance using rotations. (many students suggested to use rotations to rebalance the tree, but this is not an easy solution, because unlike the AVL-tree case, where the balance invariant holds at all but one node, here we start with a tree with many out-of-balance nodes.)

Traverse the tree in-order to extract a sorted list of elements. Now build a new tree by selecting the middle element as the root and splitting the list in half. The children of the root are the middle elements of each of the two sub-lists. Continue this procedure until all the nodes are in the tree. (You effectively place the nodes in the tree in the order that the nodes would be traversed in a binary search).

(c) What is the worst-case running time (in Θ notation) for a sequence of n INSERT operations in Ben's scheme? Assume the SRT is initially empty.

Solution: $\Theta(n^2)$. The rebalancing operations add up to $\Theta(n)$, but the cost of the last $n/4$ insertions could be as high as $\Theta(n)$ each, using the analysis in part (a). The first $3n/4$ cost $O(n)$ each, so this gives the total of $\Theta(n^2)$.

Problem 6. Ben's List Matcher [25 points] (3 parts)

Ben got tired of dealing with SRTs and decides to build a matcher for lists of numbers. Ben's matcher takes two lists of numbers and decides the lists are equal if and only if they contain exactly the same set of numbers. For example, `[1, 11, 13, 27]` is equal to `[11, 1, 27, 13]`, but not equal to `[1, 11, 13, 27, 2]` or `[1, 11, 27]`. Assume the lists do not contain multiple instances of the same number.

Ben implements his function in Python as follows:

```
def listcmp(list1, list2):
    for num in list1:
        if num not in list2:
            return False
        else:
            # Remove the first occurrence of num from list2
            list2.remove(num)
    # Return True iff list2 is now empty
    return list2 == []
```

(The python function `L.remove(x)` for a list `L` is a mutator that removes the first occurrence of `x` from `L`. Its implementation scans the list `L` from the beginning until it finds `x`.)

(a) Let n be the length of `list1` and m be the length of `list2`. What is the worst-case running time of Ben's implementation? Justify your answer.

Solution: $\Theta(nm)$. There are n iterations of the `for` loop, and each iteration removes an element from a Python list. In the first iteration the list is of size m, then $m - 1$, and so on. If $m > n$, then at least half the iterations cost $\Theta(m)$, when `list2` still has $m/2$ or more elements.

(b) Louis Reasoner suggests Ben implement his function with an AVL tree. In Louis' implementation, the elements of `list1` are inserted into an AVL tree, then the elements of `list2` are searched for and deleted if found from the AVL tree. The procedure returns `True` if and only if every element of `list2` was found in the tree, and the tree is empty after the elements of `list2` are removed.

What is the worst-case running time of Louis' implementation? Justify your answer.

Solution: Inserting n elements into the tree costs $\Theta(n \lg n)$. If we always search for all the elements of `list2` in the tree (even if they are not in the tree), then all of these searches cost $\Theta(m \lg n)$., so the total is $\Theta((m + n) \lg n)$. If we assume that the algorithm stops and returns false as soon as an element of `list2` is not in the tree, then the total cost is $\Theta(n \lg n)$, because at most n searches can be successful. Also note that the cost is $\Omega(n \lg n)$, because the first $n/2$ successful searches operate on a tree of height $\Theta(\lg n)$ in spite of the deletions.

(c) Assume the elements of `list1` and `list2` are all in $\{1, 2, ..., k\}$. Describe (in English) a solution with a $\Theta(k + n)$ worst-case running time.

Solution: if the lists do not have the same length, return false. Otherwise use counting sort to sort both and compare. If the sorted lists are not identical, return false, else return true. An even simpler algorithm is to use an array of k counters to count occurrences in `list1`, then scan `list2` and decrement the appropriate counters. If all the counters are zero at the end, return true, otherwise return false. This is again done after ensuring that the lists have the same size. (If we run counting sort or the simpler algorithm on both lists without first checking that the have the same lengths, the running time is $\Theta(n + m + k)$, which is not what the question asks for.)

Problem 7. Dynamic Medians [25 points] (3 parts)

Marianne Midling needs a data structure "DM" for maintaining a set S of numbers, supporting the following operations:

- Create an empty set S
- Add a new given number x to S
- Return a median of S. (Note that if S has even size, there are two medians; either may be returned. A median of a set of n distinct elements is larger than or equal to exactly $\lfloor (n+1)/2 \rfloor$ or $\lceil (n+1)/2 \rceil$ elements.)

(Assume no duplicates are added to S.)

Marianne proposes to implement this "dynamic median" data structure DM using a max-heap A and a min-heap B, such that every element in A is less than every element in B, and the size of A equals the size of B, or is one less.

To return a median of S, she proposes to return the minimum element of B.

(a) Argue that this is correct (i.e., that a median is returned).

Solution: This problem is on Problem Set 3 in Spring 2009; you'll have to solve it yourself. (Sorry!)

(b) Explain how to add a new number y to this data structure, while maintaining the relevant properties.

Solution: This problem is on Problem Set 3 in Spring 2009; you'll have to solve it yourself. (Sorry!)

(c) How much time does your solution to (b) take, in the worst case?

Solution: This problem is on Problem Set 3 in Spring 2009; you'll have to solve it yourself. (Sorry!)

SCRATCH PAPER

SCRATCH PAPER