Quiz 1 Solutions

Problem 1. Asymptotic Notation [25 points] (5 parts)

State whether each statement below is **True** or **False**. You must briefly justify all your answers to receive full credit.

(a) If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $h(n) = \Theta(f(n))$

Solution: True. Θ is transitive.

(b) If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $h(n) = \Omega(f(n))$

Solution: True. O is transitive, and $h(n) = \Omega(f(n))$ is the same as $f(n) = O(h(n))$

(c) If $f(n) = O(g(n))$ and $g(n) = O(f(n))$ then $f(n) = g(n)$

Solution: False: $f(n) = n$ and $g(n) = n + 1$.

(d) $\dfrac{n}{100} = \Omega(n)$

Solution: True. $\frac{n}{100} < c * n$ for $c = \frac{1}{200}$.

(e) $f(n) = \Theta(n^2)$, where $f(n)$ is defined to be the running time of the program `A(n)`:

```
def A(n):
  atuple = tuple(range(0, n)) # a tuple is an immutable version of a
                              # list, so we can hash it
  S = set()
  for i in range(0, n):
    for j in range(i+1, n):
      S.add(atuple[i:j])   # add tuple (i,...,j-1) to set S
```

Solution: False: Inside the two for loops, both slicing and hashing take linear time.

Problem 2. Weight-Balanced Binary Search Trees [20 points] (2 parts)

Recall from class our definition of a *weight-balanced binary search tree*: it maintains the weight balance of each node x, such that if the weight of x is $w(x)$, the weight of each child is at least $\alpha \cdot w(x)$, where $\alpha = 0.29$. (The *weight* of x is one more than the number of nodes in the subtree rooted at x.) This property is maintained when a node is inserted by performing rotations and double rotations going up the insertion path to fix any nodes that are not weight balanced.

Let T be an arbitrary weight-balanced tree with n nodes, for some $n \geq 1$.

(a) Show that there is a leaf node in T that could be removed without unbalancing any subtrees of T (including T itself).

Solution: Walk down T from the root, choosing the larger subtree at each step, or choosing one at random if the subtrees are equal in size, until a leaf is reached. Then that leaf can be removed. No nodes will become unbalanced: if the subtrees differed in weight, they became more balanced (α moves toward 0.5), and if they had the same weight k, the new balance is $\alpha = \frac{k-1}{2k-1} \geq 1/3 > 0.29$, since $k \geq 2$.

(b) Argue that T could have been created by a sequence of n insertions in such a way that *no* rotations were ever performed as T was built; i.e. that the growing tree was *always* weight-balanced.

Solution: Remove the nodes from T, one at a time. Since a leaf can always be removed from T (as long as it is nonempty), leaving another weight-balanced BST, we can repeat this process until T is empty. After each step, note that we can re-add the removed node, and no rotations need to be performed. Then, just insert the nodes into an empty weight-balanced BST in reverse order from the removals.

Note that this problem asks for a reconstruction of the exact tree T, not just any weight-balanced BST on the same keys.

Problem 3. Linked List Equivalence [15 points] (1 parts)

Let S and T be two sets of numbers, represented as unordered linked lists of distinct numbers. All you have are pointers to the heads of the lists, but **you do not know the list lengths**. Describe an $O(\min\{|S|, |T|\})$-expected-time algorithm to determine whether $S = T$. You may assume that any operation on one or two numbers can be performed in constant time.

Solution: First, check that both sets are the same size. If they are not, then they cannot be equal. To do this check in $O(min\{|S|, |T|\})$ time, just iterate over both lists in parallel. That is, advance one step in S and one step in T. If both lists end, the lengths are the same. If one list ends before the other, they have different lengths.

If both lists are the same size, then we want to check whether the elements are the same. We create a hash table of size $\Theta(|S|)$ using universal hashing with chaining. We iterate over S, adding each element from S to the hash table. Then we iterate over T. For each element $x \in T$, we check whether x belongs to the hash table (that is, whether it is also in S). If not, then we return that the sets are not identical. If so, then continue iterating over T.

Any sequence of $|S| = |T|$ `Insert` and `Search` operations in the table take $O(|S|)$ time in expectation (see CLRS p.234), so the total runtime is $O(\min\{|S|, |T|\})$ in expectation.

Problem 4. Hash Table Analysis [15 points] (2 parts)

You are given a hash table with n keys and m slots, with the simple uniform hashing assumption (each key is equally likely to be hashed into each slot). Collisions are resolved by chaining.

(a) What is the probability that the first slot ends up empty?

Solution: Independently, each key has a $1/m$ probability of hashing into the first slot, or $(m-1)/m$ probability of not hashing into the first slot. Thus, the probability that no key hashes into the first slot is

$$\left(\frac{m-1}{m}\right)^n.$$

(b) What is the expected number of slots that end up not being empty?

Solution: Let X_i be the event that slot i is nonempty. Since $X_i = 1$ when slot i is nonempty and is 0 otherwise, $E[X_i] = \Pr(X_i = 1)$ is the probability of slot i being nonempty.

The number of nonempty slots is $\sum X_i$. By linearity of expectation, the expected number of nonempty slots is $E[\sum X_i] = \sum E[X_i]$.

From part (a), the probability that the first slot is nonempty, or $E[X_1]$, is then $1 - (\frac{m-1}{m})^n$, and is the same for all slots. Thus, the expected number of nonempty slots is

$$m\left(1 - \left(\frac{m-1}{m}\right)^n\right).$$

Problem 5. Dynamic Programming [25 points] (1 parts)

You are given a sequence of n numbers (positive or negative):

$$x_1, x_2, \ldots, x_n$$

Your job is to select a subset of these numbers of maximum total sum, subject to the constraint that you can't select two elements that are adjacent (that is, if you pick x_i then you cannot pick either x_{i-1} or x_{i+1}).

Explain how you can find, in time polynomial in n, the subset of maximum total sum.

Solution: Let sum_i be the maximum sum of the numbers $x_1, x_2, \ldots, x_i$ given the adjacency constraint.

$$\begin{aligned} sum_0 &= 0 \\ sum_1 &= \max(0, x_1) \\ sum_i &= \max(sum_{i-2} + x_i, sum_{i-1}) \end{aligned}$$

This last step works because either we include x_i, in which case we also want to include the best solution on up to $i-2$, or we don't include x_i, in which case we can just use the best solution on $i-1$.

Our final answer is then just sum_n.

To calculate the set that gives the max sum, we could simply keep pointers back from i to either $i-1$ or $i-2$ depending on which one was bigger (or we could go back and check which was bigger). We follow those pointers, including appropriate numbers.